DATA I/O

Digitized by the Internet Archive
in 2025

https://archive.org/details/manualsplus_10103

DATA I/O

ABEL™ 2.0

JANUARY 1986

Manual Assembly: 981-0118-001

This software product is copyrighted and all rights are reserved by Data I/O Corporation. The distribution and sale of this product are intended for the use of the original purchaser only and for use only on the computer system specified. Lawful users of this product are hereby licensed only to read the programs on the disks, cassettes, or tapes, from their medium into the memory of a computer solely for the purpose of executing them. Unauthorized copying, duplicating, selling, or otherwise distributing this product is a violation of the law, except that the ABEL™ tutorial disk may be copied and distributed without further permission from, or payment to, Data I/O Corporation.

This manual (including the User's Guide, Language Reference, and Applications Guide) is copyrighted and all rights are reserved. This document may not, in whole or part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from Data I/O Corporation.

IBM is a registered trademark of International Business Machines Corporation, MS-DOS is a trademark of Microsoft Corporation, PAL is a registered trademark of Monolithic Memories Inc., PC-WRITE is a trademark of Quicksoft, Quicksoft is a trademark of Quicksoft, Signetics is a trademark of Signetics Corporation, Wordstar is a trademark of MicroPro International Corporation.

Compatibility with Available Hardware and Software

Data I/O Corporation does not warrant that its ABEL™ software package will function properly in every hardware and software environment. Data I/O Corporation warrants only that the ABEL™ software package functions properly in combination with the hardware and software described in this manual and associated documentation.

In particular, the software may not be operable in combinations with print-spooling programs or with modified versions of the operating system. The ABEL™ software does not compensate for incompatibilities in operating system revisions or versions. Running the ABEL™ software under various revisions or versions, or switching between versions at different times may result in drastic data loss or other alteration or disruption of data.

All rights reserved. Printed in the United States of America
Copyright 1983, 1984, 1986 Data I/O Corporation
Text Part No. 090-0037-002

TABLE OF CONTENTS

USER'S GUIDE

SECTION 1. INTRODUCTION

1.1 MORE ABOUT ABEL FEATURES 1-6
1.2 SYSTEM REQUIREMENTS 1-7
1.3 CONTENTS OF THE ABEL PACKAGE 1-8
1.4 DOCUMENTATION 1-10
1.5 DESIGN EXAMPLE FILES 1-12
1.6 NOTATION CONVENTIONS 1-15
1.7 SOFTWARE UPDATE SERVICE 1-17
1.8 FIELD CUSTOMER SUPPORT 1-17
1.9 LICENSE AGREEMENT AND WARRANTY 1-18
1.10 ORDERING 1-22

SECTION 2. INSTALLATION

SECTION 3. ABEL SOURCE FILES

3.1 ELEMENTS OF THE ABEL SOURCE FILE 3-2
3.2 EXAMINING AN EXAMPLE ABEL SOURCE FILE 3-6
3.2.1 Purpose of the Address Decoder 3-7
3.2.2 The MODULE Statement 3-7
3.2.3 The FLAG Statement 3-8
3.2.4 The TITLE Statement 3-8
3.2.5 The DEVICE Declaration Statement 3-9
3.2.6 PIN and NODE Declarations 3-9
3.2.7 CONSTANT Declarations 3-10
3.2.8 EQUATIONS Statements 3-11
3.2.9 Test Vectors 3-12
3.3 PROCESSING AN ABEL SOURCE FILE 3-13
3.3.1 Entering the Command Line 3-14
3.3.2 New Files Created 3-16
3.4 DOWNLOADING THE PROGRAMMER LOAD FILE . . 3-17

SECTION 4. THE ABEL LANGUAGE PROCESSOR

4.1 ABEL BATCH PROCESSING 4-3
4.1.1 Device Library 4-4
4.1.2 Generated Output Files 4-5
4.1.3 Creating Your Own Batch or Command Files 4-6
4.2 PARSE 4-7
4.2.1 PARSE Listing File 4-11
4.3 TRANSFOR 4-13
4.4 REDUCE 4-15
4.5 FUSEMAP 4-18
4.6 SIMULATE 4-22
4.6.1 SIMULATE Output File 4-28
4.6.2 Simulation for Designs With Feedback 4-32
4.7 DOCUMENT 4-39
4.7.1 DOCUMENT Output File 4-42

SECTION 5. PROGRAMMER LOAD FILE TRANSFER

5.1 DOWNLOADING TO A MODEL 29 WITH LOGICPAK 5-2
5.2 DOWNLOADING TO A MODEL 60 PROGRAMMER 5-4
5.3 PROM DOWNLOAD (MODEL 29 WITH UNIPAK 2) . . . 5-5

SECTION 6. ABEL UTILITIES

6.1 TOABEL, PALASM TO ABEL CONVERTER 6-1
6.2 IFLDOC 6-2
6.3 DEVLIB, DEVICE LIBRARY MANAGER 6-4

APPENDIX A. ERROR MESSAGES

A.1 GENERAL ERROR MESSAGES A-2
A.1.1 Command Line Errors A-2
A.1.2 Fatal Errors A-3
A.1.3 Intermediate File Errors A-5
A.1.4 Logical Errors A-7
A.1.5 Preprocessor Errors A-10
A.1.6 Syntax Errors A-11
A.1.7 Device File/Internal Errors A-14
A.2 NON-FATAL SIMULATION ERRORS A-15
A.3 TOABEL ERROR MESSAGES A-16

A.4 IFLDOC ERRORS A-17
A.4.1 Command Line Errors A-17
A.4.2 Diagnostic Errors A-18
A.4.3 JEDEC Input File Errors A-18
A.4.4 IFLDOC Warning Messages A-19

APPENDIX B. JEDEC FORMAT

B.1 BNF RULES AND STANDARD DEFINITIONS B-1
B.2 TRANSMISSION PROTOCOL B-4
B.3 DESIGN SPECIFICATION B-5
B.4 TRANSMISSION CHECKSUM B-5
B.5 FIELDS B-6
B.5.1 Device Field B-7
B.5.2 Fuse Information Fields B-8
B.5.3 Structured Functional Test Information B-10
B.5.4 Optional Information Fields B-12
B.6 OTHER RULES B-14
B.6.1 Transportability B-14
B.6.2 Variation From Present JEDEC Standard B-15
B.6.3 Variations for 303A LOGICPAK V01 . . . . . B-15

APPENDIX C. DEVICE NODES

LANGUAGE REFERENCE

SECTION 1. INTRODUCTION

1.1 OTHER ABEL™ MANUALS 1-1
1.2 NOTATION CONVENTIONS 1-2

SECTION 2. LANGUAGE ELEMENTS

2.1 BASIC SYNTAX 2-1
2.2 VALID ASCII CHARACTERS 2-2
2.3 IDENTIFIERS 2-3
2.3.1 Reserved Identifiers 2-4
2.3.2 Choosing Identifiers 2-5
2.4 STRINGS 2-6
2.5 COMMENTS 2-7
2.6 NUMBERS 2-8
2.7 SPECIAL CONSTANT VALUES 2-10

2.8 OPERATORS, EXPRESSIONS, AND EQUATIONS 2-11
2.8.1 Logical Operators 2-11
2.8.2 Arithmetic Operators 2-12
2.8.3 Relational Operators 2-13
2.8.4 Assignment Operators 2-15
2.8.5 Expressions 2-16
2.8.6 Equations 2-18
2.9 SETS 2-20
2.9.1 Set Operations 2-21
2.9.2 Set Assignment and Comparison 2-24
2.9.3 Restrictions on Sets 2-25
2.10 BLOCKS 2-26
2.11 ARGUMENTS AND ARGUMENT SUBSTITUTION 2-27

SECTION 3. STRUCTURE

3.1 BASIC STRUCTURE 3-1
3.2 MODULE STATEMENT AND STRUCTURE 3-3
3.3 FLAG STATEMENT 3-5
3.4 TITLE STATEMENT 3-6
3.5 DECLARATIONS 3-7
3.5.1 Device Declaration Statement 3-7
3.5.2 Pin Declaration Statement 3-8
3.5.3 Node Declaration Statement 3-10
3.5.4 Constant Declaration Statement 3-12
3.5.5 Macro Declaration Statement and Macro Expansion 3-14
3.5.6 ISTYPE Statement 3-17
3.6 EQUATIONS STATEMENT 3-21
3.7 TRUTH TABLES 3-22
3.7.1 Truth Table Header Syntax 3-22
3.7.2 Truth Table Format 3-24
3.8 STATE DIAGRAMS 3-26
3.8.1 STATE_DIAGRAM Statement 3-26
3.8.2 IF-THEN-ELSE Statement 3-29
3.8.3 CASE Statement 3-31
3.8.4 GOTO Statement 3-32
3.9 FUSES SECTION 3-35
3.10 TEST VECTORS 3-36
3.10.1 Test Vector Table Format 3-37

SECTION 4. DIRECTIVES

4.1 @ALTERNATE Directive 4-2
4.2 @CONST (Constant) Directive 4-3
4.3 @EXIT Directive 4-3
4.4 @EXPR (Expression) Directive 4-4
4.5 @IF Directive 4-4
4.6 @IFB (If Blank) Directive 4-5
4.7 @IFDEF (If Defined) Directive 4-5
4.8 @IFIDEN (If Identical) Directive 4-6
4.9 @IFNB (If Not Blank) Directive 4-6
4.10 @IFNDEF (If Not Defined) Directive 4-7
4.11 @IFNIDEN (If Not Identical) Directive 4-7
4.12 @INCLUDE Directive 4-8
4.13 @IRP (Indefinite Repeat) Directive 4-9
4.14 @IRPC (Indefinite Repeat, Character) Directive . 4-10
4.15 @MESSAGE Directive 4-11
4.16 @PAGE Directive 4-11
4.17 @RADIX Directive 4-12
4.18 @REPEAT Directive 4-13
4.19 @STANDARD Directive 4-13

APPENDIX A. SYNTAX DIAGRAMS

A.1 How to Read Syntax Diagrams A-1
A.2 ABEL™ Syntax Diagrams A-3

APPENDIX B. ASCII TABLES

APPENDIX C. SET OPERATIONS

APPLICATIONS GUIDE

SECTION 1. INTRODUCTION

1.1 DESIGN EXAMPLE FILES 1-2
1.2 OTHER ABEL™ MANUALS 1-5
1.3 NOTATION CONVENTIONS 1-5

SECTION 2. DESIGN EXAMPLES

2.1 6809 MEMORY ADDRESS DECODER 2-2
2.2 12 TO 4 MULTIPLEXER 2-5
2.3 1 TO 8 DEMULTIPLEXER 2-8
2.4 4-BIT COUNTER/MULTIPLEXER 2-11
2.5 DECIMAL UP/DOWN COUNTER 2-17
2.6 8-BIT BARREL SHIFTER 2-22
2.7 7-SEGMENT DISPLAY DECODER 2-26
2.8 4-BIT COMPARATOR 2-30
2.9 BIDIRECTIONAL TRI-STATE BUFFER 2-36
2.9.1 Design Specification 2-36
2.9.2 Design Method 2-37

SECTION 3. DESIGN CONSIDERATIONS

3.1 USING STATE MACHINES 3-1
3.1.1 Use Identifiers Rather Than Numbers for States 3-2
3.1.2 "Power On" Register States 3-4
3.1.3 Design With the Registers, Not With the Outputs 3-5
3.1.4 Unsatisfied Transition Conditions Clear the State Register 3-5
3.1.5 Number Adjacent States for One-Bit Changes 3-6
3.1.6 Use State Register Outputs to Identify States 3-7
3.1.7 Combine the Use of Equations and State Diagrams 3-8
3.2 MULTIPLE ASSIGNMENTS TO THE SAME SIGNAL 3-9
3.3 HAZARD-FREE LOGIC DESIGN AND "REDUCE" 3-12
3.4 PASSING ARGUMENTS FROM THE COMMAND LINE 3-14
3.5 EFFECT OF POLARITY ON REDUCTION SPEED 3-18

SECTION 4. SIMULATION

4.1 TEST VECTORS AND SIMULATION 4-2
4.2 TRACE LEVELS AND BREAKPOINTS 4-2
4.3 DEBUGGING STATE MACHINES 4-4
4.4 MULTIPLE TEST VECTOR TABLES 4-5
4.5 USING MACROS AND DIRECTIVES TO CREATE TEST VECTORS 4-6
4.6 DON'T CARES IN SIMULATION 4-11
4.7 PRESET AND PRELOAD REGISTERS 4-13
4.7.1 Special Preset Considerations 4-14
4.7.2 TTL Preload 4-17
4.7.3 Supervoltage Preload 4-19
4.7.4 Preset/Reset Controlled by Product Term . . . 4-22
4.8 ASYNCHRONOUS CIRCUITS 4-24

SECTION 5. USING ADVANCED DEVICE FEATURES

5.1 OUTPUT ENABLES 5-1
5.1.1 Pin Controlled Output Enable 5-2
5.1.2 Term Controlled Output Enable 5-3
5.1.3 Configurable Output Enable 5-4
5.2 OUTPUT MACRO CELL CONTROL WITH ISTYPE 5-6
5.2.1 Controlling Macro Cell Polarity 5-6
5.2.2 Controlling Macro Cell Feedback Point . . . 5-8
5.2.3 Controlling Macro Cell Type 5-10
5.2.4 Controlling Register Type 5-10
5.3 NODES 5-11
5.3.1 Using Node Numbers 5-13
5.3.2 Using Shorehand Notation for Nodes 5-15
5.4 USING FLIP-FLOPS (MACRO CELL DEVICES) 5-17
5.4.1 The 82S159 Logic Diagram 5-18
5.4.2 Examination of the Source File 5-20
5.4.3 Programming the Flip-Flop Mode Control Fuses 5-22
5.4.4 Combining Equations 5-25
5.4.5 Specifying the Flip-Flop Inputs 5-26
5.4.6 Test Vectors 5-29
5.5 USING TRANSITION EQUATIONS 5-31

SECTION 6. ADVANCED DESIGNS

6.1 BLACKJACK MACHINE 6-2
6.2 6809 SYSTEM CONTROLLER 6-21

USER'S GUIDE

DATA I/O

TABLE OF CONTENTS

SECTION 1. INTRODUCTION

1.1 MORE ABOUT ABEL FEATURES 1-6
1.2 SYSTEM REQUIREMENTS 1-7
1.3 CONTENTS OF THE ABEL PACKAGE 1-8
1.4 DOCUMENTATION 1-10
1.5 DESIGN EXAMPLE FILES 1-12
1.6 NOTATION CONVENTIONS 1-15
1.7 SOFTWARE UPDATE SERVICE 1-17
1.8 FIELD CUSTOMER SUPPORT 1-17
1.9 LICENSE AGREEMENT AND WARRANTY 1-18
1.10 ORDERING 1-22

SECTION 2. INSTALLATION

SECTION 3. ABEL SOURCE FILES

3.1 ELEMENTS OF THE ABEL SOURCE FILE 3-2
3.2 EXAMINING AN EXAMPLE ABEL SOURCE FILE 3-6
3.2.1 Purpose of the Address Decoder 3-7
3.2.2 The MODULE Statement 3-7
3.2.3 The FLAG Statement 3-8
3.2.4 The TITLE Statement 3-8
3.2.5 The DEVICE Declaration Statement 3-9
3.2.6 PIN and NODE Declarations 3-9
3.2.7 CONSTANT Declarations 3-10
3.2.8 EQUATIONS Statements 3-11
3.2.9 Test Vectors 3-12
3.3 PROCESSING AN ABEL SOURCE FILE 3-13
3.3.1 Entering the Command Line 3-14
3.3.2 New Files Created 3-16
3.4 DOWNLOADING THE PROGRAMMER LOAD FILE . . 3-17

SECTION 4. THE ABEL LANGUAGE PROCESSOR

4.1 ABEL BATCH PROCESSING 4-3
4.1.1 Device Library 4-4
4.1.2 Generated Output Files 4-5
4.1.3 Creating Your Own Batch or Command Files 4-6
4.2. PARSE 4-7
4.2.1 PARSE Listing File 4-11

4.3 TRANSFOR 4-13
4.4 REDUCE 4-15
4.5 FUSEMAP 4-18
4.6 SIMULATE 4-22
4.6.1 SIMULATE Output File 4-28
4.6.2 Simulation for Designs With Feedback 4-32
4.7 DOCUMENT 4-39
4.7.1 DOCUMENT Output File 4-42

SECTION 5. PROGRAMMER LOAD FILE TRANSFER
5.1 DOWNLOADING TO A MODEL 29 WITH LOGICPAK 5-2
5.2 DOWNLOADING TO A MODEL 60 PROGRAMMER 5-4
5.3 PROM DOWNLOAD (MODEL 29 WITH UNIPAK 2) . . . 5-5

SECTION 6. ABEL UTILITIES
6.1 TOABEL, PALASM TO ABEL CONVERTER 6-1
6.2 IFLDOC 6-2
6.3 DEVLIB, DEVICE LIBRARY MANAGER 6-4

APPENDIX A. ERROR MESSAGES
A.1 GENERAL ERROR MESSAGES A-2
A.1.1 Command Line Errors A-2
A.1.2 Fatal Errors A-3
A.1.3 Intermediate File Errors A-5
A.1.4 Logical Errors A-7
A.1.5 Preprocessor Errors A-10
A.1.6 Syntax Errors A-11
A.1.7 Device File/Internal Errors A-14
A.2 NON-FATAL SIMULATION ERRORS A-15
A.3 TOABEL ERROR MESSAGES A-16
A.4 IFLDOC ERRORS A-17
A.4.1 Command Line Errors A-17
A.4.2 Diagnostic Errors A-18
A.4.3 JEDEC Input File Errors A-18
A.4.4 IFLDOC Warning Messages A-19

APPENDIX B. JEDEC FORMAT

B.1 BNF RULES AND STANDARD DEFINITIONS B-1
B.2 TRANSMISSION PROTOCOL B-4
B.3 DESIGN SPECIFICATION B-5
B.4 TRANSMISSION CHECKSUM B-5
B.5 FIELDS B-6
B.5.1 Device Field B-7
B.5.2 Fuse Information Fields B-8
B.5.3 Structured Functional Test Information B-10
B.5.4 Optional Information Fields B-12
B.6 OTHER RULES B-14
B.6.1 Transportability B-14
B.6.2 Variation From Present JEDEC Standard B-15
B.6.3 Variations for 303A LOGICPAK V01 . . . . . B-15

APPENDIX C. DEVICE NODES

FIGURES

1-1 ABEL, A Logic Design Tool 1-2
1-2 Logic Design Steps with ABEL 1-5
3-1 Source File Template 3-4
3-2 Block Diagram: 6809 Memory Address Decoder . 3-5
4-1 Processing Flow of the Language Processor 4-2
4-2 Simulation Output for M6809A.ABL 4-28
4-3 Test Vectors Used to Create Simulation Errors 4-29
4-4 Trace Level 3 Simulation Output 4-30
4-5 Trace Level 4 Simulation Output 4-31
4-6 Synchronous Feedback Circuit 4-32
4-7 Asynchronous Feedback Circuit 4-36
5-1 Cable Configuration for Transfer Between an IBM-PC/XT and a Data I/O Programmer 5-3
5-2 Cable Configuration for Transfer Between an IBM-AT and a Data I/O Programmer 5-3
6-1 Sample Output File from IFLDOC 6-3
B-1 Example of a JEDEC Format Transmission File B-4
B-2 8-Bit Words Formed from Fuse States for Checksum B-9
B-3 'Kernel' Transmission File B-15

LISTINGS

3-1 Source File Describing an Address Decoder 3-5
3-2 Console Display for the Processing Operation . . . 3-15
4-1 PARSE Listing File with Errors from M6809ERR.ABL . . 4-12
4-2 Corrected Source File, M6809A.ABL 4-12
4-3 Trace Level 1 Simulation Output 4-29
4-4 Source File: Synchronous Feedback Circuit 4-33
4-5 Simulation Output, Trace Level 1: Synchronous Feedback Circuit 4-33
4-6 Simulation Output, Trace Level 2: Synchronous Feedback Circuit 4-34
4-7 Simulation Output, Trace Level 3: Synchronous Feedback Circuit 4-34
4-8 Source File: Asynchronous Feedback Circuit 4-36
4-9 Simulation Output, Trace Level 1: Asynchronous Feedback Circuit 4-37
4-10 Simulation Output, Trace Level 2: Asynchronous Feedback Circuit 4-37
4-11 Simulation Output, Trace Level 3: Asynchronous Feedback Circuit 4-38
4-12 Documentation Output for M6809A.ABL 4-42

TABLES

1-1 Files Supplied with ABEL 1-9
1-2 Design Examples Supplied with ABEL 1-14
1-3 Notation Conventions 1-15
4-1 Data Translation Format Codes and File Extensions 4-21
B-1 Field Identifiers B-7
B-2 Test Conditions B-11
C-1 Nodes Supported By ABEL C-1

1 INTRODUCTION

ABEL™ is a complete logic design tool that lets you easily describe and implement programmable logic designs in PALs, IFLs, and PROMs. ABEL™ consists of a special-purpose, high-level language that is used to describe logic designs, and a language processor that converts logic descriptions to programmer load files. Programmer load files contain the information necessary to program and test programmable logic devices. Figure 1-1 shows ABEL™ used with Data I/O's Model 29 Logic Programmer in the Data I/O Personal Silicon Foundry, a total programmable logic development system.

ABEL™ may be used with several other Data I/O design development tools such as:

- PLDtest™; an automatic test vector generator that allows 100% testing of programmed logic parts
- DASH/ABEL™; a schematic diagram interface that converts schematic designs to ABEL™ source files
- PROMlink ™; a program that permits control of and communication with Data I/O programmers by means of a personal computer.

Features of the ABEL™ design language are:

- Universal syntax for all programmable logic types
- High-level, structured design language
- Flexible forms for describing logic:
  - Boolean Equations
  - Truth Tables
  - State Diagrams
- Test Vectors for simulation and testing
- Time-Saving Macros and Directives

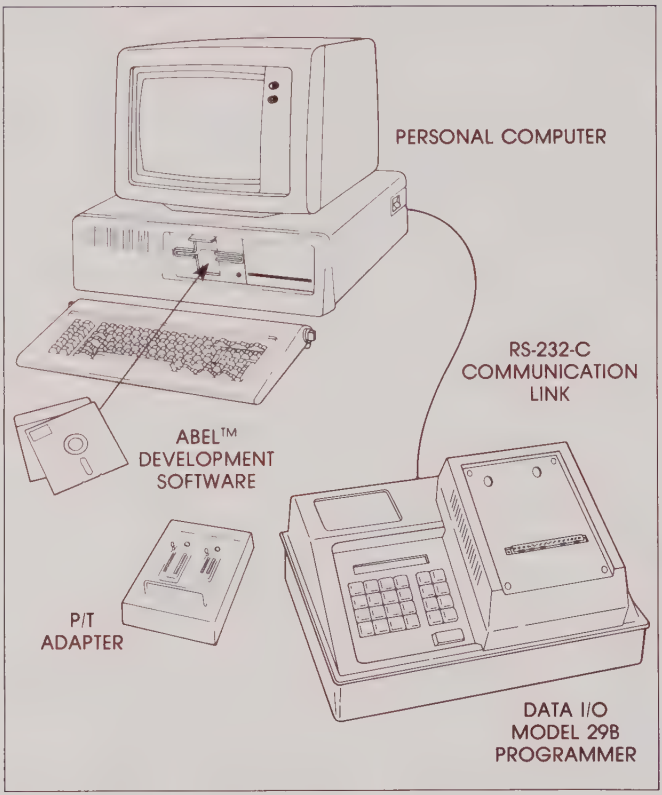


Figure 1–1. ABEL™, A Logic Design Tool

The ABEL™ language processor also has many powerful features:

- Syntax Checking
- Verification that a design can be implemented with a chosen part
- Logic Reduction
- Design Simulation
- Automatic design documentation
- Creation of programmer load files in JEDEC and PROM format

Together, the ABEL™ design language and language processor make it easy to design and test logic functions that are to be implemented with programmable logic devices. For example, you can design a three-input AND function with the inputs *A*, *B*, and *C* and the output *Y* using a truth table like this:

```
truth_table "3-input AND gate"

( [ A , B , C ] -> Y )

  [ 0,.X.,.X.]-> 0 ;
  [.X., 0,.X.] -> 0 ;
  [.X.,.X., 0] -> 0 ;
  [ 1, 1, 1] -> 1 ;
```

The ".X."s in the table indicate "don't care" conditions, and the output *Y* is set to 1 only when all three inputs equal 1. You also could have specified the output *Y* in terms of simple Boolean operators and have achieved the same result. This is done here, where "&" is the logical AND operator:

```
Y = A & B & C ;
```

ABEL™ lets you choose the type of description that is best suited to the logic being described, or the type of description you feel most comfortable with. And, in most cases, the same description can be used for many different devices simply by changing the device specification. ABEL™ enters the design process in a way that reduces errors and saves time. You can think about designs in a logical, functional way, describe them in that fashion, and then test your design to see that it operates as expected, all without worrying about which fuses should be blown or left intact.

Figure 1–2 shows the logic design process and the role ABEL™ takes in it. Beginning with the design concept, the designer creates the ABEL™ source file required by the language processor in order for it to generate the programmer load file. The source file is written by you and contains a complete description of your logic design. You can create the source file manually be means of a text editor (or word processor) that generates ASCII files, or you can use DASH–ABEL™ to convert a DASH-generated schematic of the design to an ABEL™ source file

The source file is presented to the ABEL™ language processor which performs several functions to produce a programmer load file (in JEDEC format) and design documentation. The first ABEL™ function, Parse, checks the syntax of the source file and flags any errors. Transform converts the logic description to an intermediate form. Reduce performs logic reduction, and Fusemap creates the programmer load file. The programmer load file can then be downloaded to the logic programmer to program parts, or can be first transmitted to PLDtest, an automatic test vector generator. The Simulate function tests the design of the part against your test vectors contained in the source file and reports any functional failure of the design. The Document function generates a listing of the source file, a drawing of the logic device pin assignments, and a listing of the programmer load file.

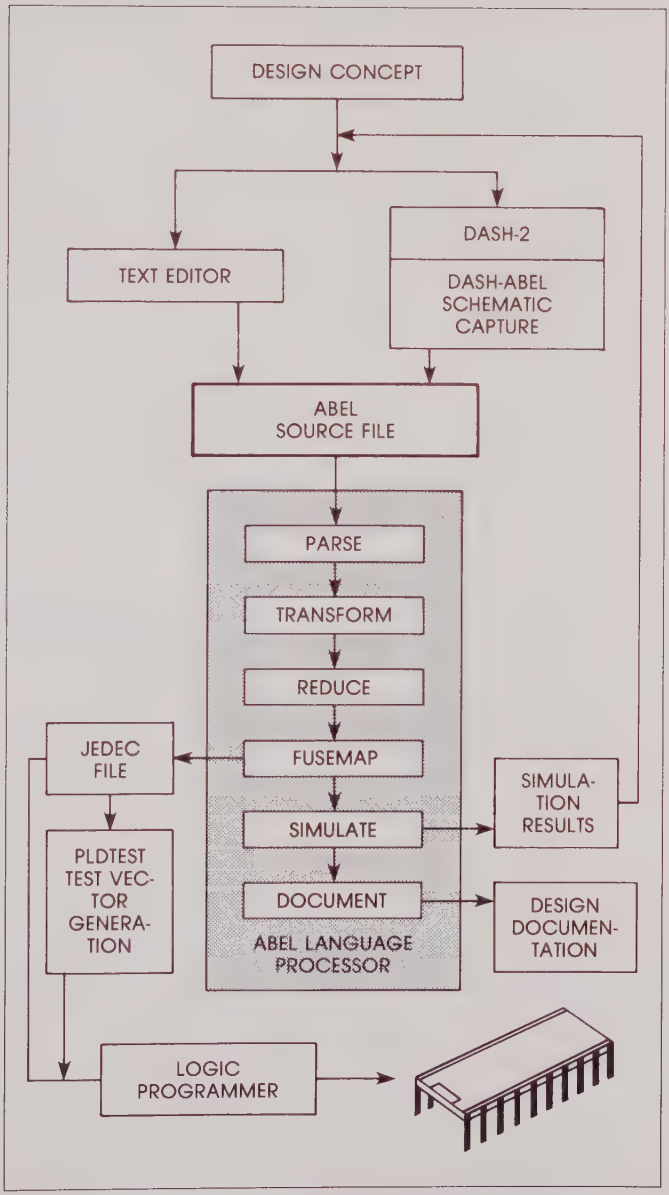


Figure 1–2. Logic Design Steps with ABEL™

Design Checking

The language processor checks your logic design for correct language syntax and explicitly tells you where an error occurs and what the error is. The language processor also checks your design to see if it can be implemented on the chosen device. For example, if a device input pin is used as an output in an equation, the language processor detects and reports the error.

Logic Reduction

The language processor reduces your logic design to a near minimal form, so that you do not have to perform the tedious task of logic reduction by traditional methods such as Karnaugh maps. You may choose different levels of reduction based on the design and the device.

Simulation

Simulation of a design is performed after a logic design has been reduced and converted to a programmer load file. The simulation facility uses device characteristics, a fuse map, and test vectors to simulate the actual operation of the device. The fuse map and test vectors used for simulation are the same as those that will be used to program and test the real device.

Functional Device Testing

If test vectors are specified in a source file, the programmer load file created by the language processor contains these vectors in a form that can be used to test a programmed device with a logic programmer.

Standard JEDEC Format Programmer Load File

The standard programmer load file created by the language processor conforms with the JEDEC standard No.3 for data transfer to logic programmers. JEDEC format files are used to transfer PAL and IFL designs to the logic programmer. Other formats for PROM programmers are supported.

SYSTEM REQUIREMENTS 1.2

ABEL™ presently runs on the following computers and operating systems. Versions for additional systems are under development.

- IBM/AT/XT™ and MS-DOS™ compatibles
- VAX/VMS™
- VAX/Unix™
- Sun
- Valid™
- Apollo/Mentor

The configuration information and installation instructions for ABEL™ differ for each type of system. To install ABEL™ in your particular system, refer to the Installation Guide supplied with your ABEL™ package. In addition to ABEL™, you will need an editor or word processor with which to create ABEL™ source files. This may be any editor of your choice as long as it produces a standard ASCII file.

Note: Some word processors, such as Wordstar, operate in more than one mode, and may create non-standard ASCII files that cannot be used with ABEL™. If you are using such a word processor, choose the mode of operation that creates printable ASCII files.

For downloading programmer load files to a logic programmer, you will need:

- An RS232 port and a cable to connect to the programmer. Cable and programmer configuration are discussed in the User's Guide.

CONTENTS OF THE ABEL™ PACKAGE 1.3

The ABEL™ software package consists of the following:

- Floppy disks or magnetic tape:
- Documentation consisting of:

User's Guide
Language Reference
Applications Guide
Index and References
Logic Diagrams
Installation and System-Specific Information

If any of the above items is missing, please contact your Data I/O representative. Table 1-1 lists the files residing on the disks or magnetic tape that are common to all versions of ABEL™. Additional files that accompany your particular version are listed in the Installation and System-Specific Information.

As a service to Data I/O's customers, the PC-WRITE editor/word processor by Quicksoft is included (with IBM PC/XT/AT versions only) at no charge so that you can create and edit ABEL™source files. Quicksoft is solely responsible for PC-WRITE. Data I/O assumes no responsibility for the PC-WRITE product, and questions concerning that product should be directed to Quicksoft, Seattle, WA.

Table 1-1. Files Supplied with ABEL™

Filename	Description
ABEL	Batch file for a complete ABEL™ run
PARSE	Language parser
TRANSFOR	Equation transformation program
REDUCE	Equation reduction program
FUSEMAP	Programmer load file generator
DOCUMENT	Documentation generator
IFLDOC	Program that converts JEDEC-format files to FPLA program tables for documentation purposes
TOABEL	PALASM to ABEL™ Converter
SIMULATE	Simulation facility
DEVLIB	Device library librarian
ABELLIB.DEV	Device file library
*.DEV	Device specification files (one for each supported device type)
*.ABL	Logic design examples
EXAMPLES.TXT	Describes the logic design examples included on the disk

DOCUMENTATION 1.4

The ABEL™ documentation is divided into three fully indexed manuals with a comprehensive index (all contained within this same binder), plus an Installation Guide tailored to you particular system. Each manual is described below.

ABEL™ User's Guide

This manual describes how to use the ABEL™ language processor to convert logic descriptions to programmer load files. Setup procedures are described and a sample session with the language processor is given. All processor options are described in detail and the output files are explained. The procedure for downloading programmer load files is presented.

ABEL™ Language Reference

This manual contains a complete definition of the ABEL™ design language and is intended for use as both a quick reference and a general introduction to the ABEL™ design language.

ABEL™ Applications Guide

This manual presents information and examples to help the designer use ABEL™. Specifications, design methods, and complete source files are given for typical programmable logic designs. Advanced designs are presented. Suggestions and tips concerning the use of ABEL™ are given.

Comprehensive Index

This index contains entries and cross references for all three manuals. For each entry in the index, the manual and the page within that manual where the entry can be found is listed.

Installation and System-Specific Information

This document describes how to install ABEL™ on your system in preparation for logic design. It also contains information regarding supplied software and files that can be run specifically on your system.

A set of device logic diagrams is provided. A logic diagram for each supported device is included in the set for reference purposes.

DESIGN EXAMPLE FILES 1.5

This manual shows complete logic designs described with ABEL and discusses general topics that you will find useful as you use ABEL. All the logic designs and design features discussed in this manual are contained on the Design Examples disk or tape you received with the ABEL package; and are listed in table 1-2.

You can process these design examples with ABEL, either as they stand, or with your own modifications, to create programmer load files. Many of the design examples listed in table 1-2 are described in detail within this manual, and the section reference is given for these design examples. Design examples not described in the manual, but listed in table 1-2, can be examined as the need arises.

Table 1-2 lists the design examples alphabetically by filename of the ABEL source file and contains columns for:

- THE NAME OF THE SOURCE FILE. The source file uses the .ABL file extension (e.g., ADD5 is ADD5.ABL) although ABEL does not require that the file extension be entered on the command line.
- PROGRAMMABLE LOGIC DEVICE TYPE. The type of programmable logic device type used in the design example is listed to assist you in locating meaningful design examples.
- SECTION REFERENCE. Where applicable, a reference to a specific section in the Applications Guide manual is given that provides some detailed information.
- TYPE OF EXAMPLE. A brief description of the design example.

Table 1-2. Design Examples* Supplied with ABEL

Filename (*.ABL)	Device Type	App. Guide Ref.	Type of Example
ADD5	F82S159	N/A	Macros for CARDSEL.ABL
ALARM	P16R4	N/A	Macros (state diagram)
ALTERA	EP300	5.2	Output macro cells
BARREL	P20R8	2.6	Sets
BCD7PAL	P16L8	N/A	PALASM example for TOABEL
BCD7ROM	RA5P8	2.7	Truth table
BGATES	P12H6	N/A	ABEL version of PALASM basic gates
BGEQN	F82S100	N/A	@repeat directive used w/equations
BJSTATE.TAB	N/A	N/A	Include file for CARDSEL.ABL
BUFFER	F82S153	2.9	Tri-state output control
CARDSEL	P16HD8	6.1	Blackjack machine
COMP4	P16HD8	2.8	Macros
COMP4ROM	RA8P4	N/A	Relational operators
CONTROL	P16R6	6.1	Blackjack state machine
COUNT10	P16R4	2.5	State Diagram
COUNT10A	P16R4	3.2	Equations and state diagram
COUNT4	P16R4	2.4	Equations
COUNT4A	P16R4	3.3	Multiple equations for one output
DECADE	F82S105	5.5	Transition equations, complement array
DMUX1T8	P16L8	2.3	Sets
FEEDBACK	P16HD8	N/A	Simulator example
FPLS	F82S157	N/A	Logic sequencer features
GATES	P12H6	N/A	PALASM example for TOABEL
LCUCROM	RA8P8	N/A	Macros, compiler directives

Table 1–2. Design Examples* Supplied with ABEL (Cont.)

Filename (*.ABL)	Device Type	App. Guide Ref.	Type of Example
LED1	F82S153	6.1	Truth tables for CARDSEL.ABL
LED2	F82S153	6.1	Truth tables for CARDSEL.ABL
M6809A	P14L4	2.1	Sets, relational operators
M6809B	P14L4	4.5	Macros with test vectors
M6809C	P14L4	4.5	Macros with test vectors
M6809D	P16HD8	3.5	Device type from command line
M6809ER	P14L4	N/A	Demonstrates simulation error
M6809ERR	P14L4	N/A	Demonstrates syntax error
MUX12T4	P14H4	2.2	Sets
OCTAL	P20X8	N/A	Exclusive OR equations
REF6809	P16L8	6.2	Multiple equation and vectors
REGFB	P16R4	N/A	Simulator example
SHIFT40	32R16	N/A	84-pin PAL example
TI1SHOT	P19R8	N/A	Input registers
TRAFFIC	F82S167	N/A	State machine RS flip/flops
SHIFTCNT	F82S159	5.2	Controlling JK and D flip/flops
VTI	E10P8	N/A	Programmable XOR

*This table represents a partial list of design examples available at the time of printing. To verify the design examples provided on your ABEL disk or tape, use the operating system to print the file named EXAMPLES.TXT.

NOTATION CONVENTIONS 1.6

Table 1-3 shows notation conventions used in the many definitions and syntax descriptions contained in this manual.

Table 1-3. Notation Conventions

Notation	Usage
italics	Indicates references by name to items contained in examples, figures, listings and tables.
quotation marks (")	Surround italicized items when the italicized reference contains spaces.
UPPER-CASE	In syntax descriptions and diagrams, indicates that the word or letters in upper case must be entered in full. The item can be entered in either upper-case, lower-case, or mixed-case. Indicates a keyword.
lower-case	In syntax descriptions and diagrams, indicates that a value or name is supplied by the user in place of the word or letters in lower-case. The user-supplied value can be entered in either upper-case, lower-case, or mixed-case.
square brackets [ ]	Surround optional entries that can be supplied or omitted as necessary.
ellipsis (...)	Indicates that the preceding item can be repeated as necessary.
all other punctuation	Apostrophes, exclamation points, parentheses, quotes and commas must be entered exactly as shown.

An example of italics used in text:

> An table 1–3, the word *ellipsis* refers to three periods in a row.

An example of quotation marks used in text:

> The file specification, "*m68O9a fus*", is incorrect because a space separates the filename from the extension. The correct specification is *m68O9a.fus*.

An example of upper–case, lower–case, square brackets, and ellipsis as used in a syntax description:

> PARSE [–lin__file] [–Aarg]...

The command, *PARSE*, must be entered, but it can be entered in either upper–case, lower–case, or mixed–case. An input file can be (but does not have to be) specified with the –I parameter by typing –I in either upper–case or lower–case followed by a file specification in place of *in__file*. Arguments can be supplied following the –A parameter. The –A parameter is typed in either upper–case or lower–case. An argument is supplied by the user in place of *arg*. As many arguments as desired can be entered, but each requires that a new –A parameter be used. The following would be a valid user entry:

> parse –Ap16r4 –aGND

SOFTWARE UPDATE SERVICE 1.7

The first year of software update service is included with the ABEL™ purchase that suppliies you with software updates, "bug-fixes," and application ideas. See the software update service card at the back of the manual to purchase renewal of this service.

FIELD CUSTOMER SUPPORT 1.8

Data I/O has Product Support Engineers (PSEs) who can provide you with additional assistance in using this product. The PSE can be reached by contacting the Data I/O Customer Resource Center (800-247-5700).

LICENSE AGREEMENT AND WARRANTY 1.9

The user should completely and carefully read the following terms and conditions before accepting delivery of this product. Possession of this software product indicates the user's acceptance of these terms and conditions. If the user does not agree with them, the user should promptly return the product and the purchase price will be refunded to the original purchaser of this product.

Data I/O Corporation, licensor, provides this set of software programs and licenses their use. The user assumes the responsibility for the selection of the programs to achieve the user's intended results, and for the installation, use, and results obtained form said program(s).

License

1. The user may use the licensed program on any single machine.

2. The user may copy the licensed program into any machine readable or printed form for backup or archival purposes in support of the user's use of the program on the single machine.

3. The user recognizes the proprietary nature of the licensed programs and materials and agrees to preserve and protect Data I/O Corporation's interest therein. All information relating to the licensed programs and materials provided to the user shall be retained by the user and shall not be used or disclosed except for the purpose of meeting the user's own internal use.

Term

The license is effective until terminated. The user may terminate the license at any time by destroying the licensed program and materials together with all copies. The license will also terminate if the user fails to comply with any term or condition of this agreement. The user agrees upon such termination to destroy the licensed program and materials together with all copies in whatever form. Within 30 days after termination, the user shall certify in writing that through its best efforts and to the best of its knowledge all copies, in part or in whole, of the terminated and/or discontinued licensed programs and materials relating thereto have been destroyed.

Limited Warranty

1. The ABEL™ Software programs are provided by Data I/O Corporation "AS IS" without warranty of any kind, including but not limited to, the warranties of merchantability and fitness for a particular purpose. Data I/O Corporation will not be liable to the user of the ABEL™ software programs for any damages, including lost profits, lost savings, loss of use or other incidental or consequential damages arising out of the use, or inability to use, the provided software programs, even if Data I/O Corporation has been advised of the possibility of such damages, or for any other claims brought by any other party.

2. Some states do not allow the limitation or exclusion of liability for incidental or consequential damages so the above limitation or exclusion may not apply to the user.

3. Data I/O Corporation does not warrant that the functions contained in the licensed program(s) will meet the user's requirements or that the operation of the licensed program(s) will be uninterrupted or error free.

4. Data I/O Corporation warrants the diskette(s) or magnetic tape or cassette(s) on which the programs are furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of shipment to the user as evidenced by a copy of the packing slip.

Limitation of Remedies

Data I/O Corporation's entire liability and the user's exclusive remedy shall be as follows:

1. The replacement of any diskette(s) or magnetic tape or cassette(s) not meeting Data I/O Corporation's limited warranty, and which materials are returned to Data I/O Corporation or an authorized Data I/O representative with a copy of the packing slip, or

2. If Data I/O Corporation or a representative is unable to deliver a replacement diskette(s), magnetic tape or cassette(s), which is/are free of defects in materials or workmanship, you may terminate this agreement by returning the programs and their related materials and your money will be refunded.

General

1. The user acknowledges that the user has read this agreement, understands it, and agrees to be bound by its terms and conditions. The user further agrees that it is the complete and exclusive statement of the agreement between the user and Data I/O Corporation, which agreement supersedes any proposal or prior agreement, whether oral or written, and any other communications between the user and Data I/O Corporation relating to the subject matter of this agreement.

2. This license and agreement shall be governed by the laws of the State of Washington.

3. The user may not sub-license, assign, or transfer the licensed program(s) except as expressly provided in this agreement. Any attempt to do otherwise, to sub-license, assign or transfer any of the rights, duties or obligations hereunder is void, except that Data I/O Corporation may assign this license as a part of an assignment of all software assets of its business.

4. The provisions of this agreement are severable and if any one or more of such provisions are judicially determined to be illegal or otherwise unenforceable, whether in whole or in part, those remaining provisions or portions of the license unaffected by such determination shall be binding on and enforceable by and between the licensee, user, and Data I/O Corporation.

5. In the event it becomes necessary to retain the services of an attorney to enforce any provisions of this license agreement, the non-prevailing party to such legal action and/or litigation agrees to pay the prevailing party's costs, including a reasonable attorney's fee, and court costs, if any.

ORDERING 1.10

To place an order for software, contact your Data I/O representative. Orders for shipment must include the following:

- Description of the software (see latest Data I/O price list or contact your sales representative for product numbers).
- Purchase order number
- Desired method of shipment
- Quantity of each item ordered
- Shipping and billing address of firm, including zip code
- Name of person ordering software

2 INSTALLATION

The procedure for installing ABEL™ in your system depends on the system (IBM-XT, VAX/VMS, etc.), installation procedures tailored to your system are contained in the Installation Guide. The Installation Guide accompanies this manual as part of your ABEL™ documentation. Refer to the Installation and System-Specific Information supplied with the ABEL™ package for details regarding installation of ABEL™ on your particular system.

3 ABEL™ SOURCE FILES

In order for the ABEL™ Language Processor to convert logic descriptions to programmer load files, the logic descriptions must be contained in an ASCII source file. That is, before you can create a load file that contains the fusemap of a particular logic design, you must create an ABEL™ source file that reflects that logic design. The ABEL™ source file is an ASCII text file that is written according to the requirements of the ABEL™ Language Processor. These requirements are described in detail in the Language Reference manual.

ELEMENTS OF THE ABEL™ SOURCE FILE 3.1

You can write an ABEL™ source file using any word processor or text editor that produces printable ASCII files. (Wordstar must be operated in the non-document mode in order to obtain ASCII files.) The elements of the ABEL™ source file are shown in the template in figure 3-1. Figure 3-1 shows the basic element of the ABEL™ source file, the module. A source may contain any number of modules, each of which contain a different logic description.

Figure 3-1 shows that the source file module (or complete source file in this case) is made up of several elements. These elements are further illustrated in listing 3-1, a source file for an address decoder that is to be placed in a P14L4.

The elements of the ABEL™ source file are:

MODULE STATEMENT – This names the module and also indicates whether any dummy arguments are used. Refer to section 3.2 of the Language Reference manual for additional details.

FLAG – This can be used to control processing of the source file by the language processor, and is optional. Refer to section 3.3 of the Language Reference manual for additional details.

TITLE – The title is optional and is used to describe the module. The text entered will be used as a header for the ABEL™ output files (see section 3.3.2).

DEVICE DECLARATIONS – The device declaration associates one or more device identifiers with specific programmable logic devices. Refer to section 3.5.1 of the Language Reference manual for additional details.

PIN DECLARATIONS – This defines the pins of the programmable logic device with identifiers used in the source file. Refer to section 3.5.2 the Language Reference manual for addtional details.

NODE DECLARATIONS – If nodes exist on the programmable logic device, they are defined here. Refer section 3.5.4 of the Language Reference manual for additional details.

CONSTANT DECLARATIONS – Refer to section 3.5.4 of the Language Reference manual for details.

MACRO DEFINITIONS – Refer to section 3.5.5 of the Language Reference manual for details.

The above source file elements are completed as necessary and can exist in any order, provided no symbol (identifier) is referenced before it has been defined.

EQUATIONS – Contains the Boolean equations required to describe the logic design. (You can also express the design by means of truth tables or state diagrams.) Refer to section 3.6 of the Language Reference manual for details on the Equations section of the source file.

TRUTH TABLE – Contains the truth tables required to describe the logic design. (You can also express the design by means of Boolean equations or state diagrams.) Refer to section 3.7 of the Language Reference manual for details on the Truth Table section of the source file.

STATE DIAGRAM – Contains the state diagrams required to describe the logic design. (You can also express the design by means of Boolean equations or truth tables.) Refer to section 3.8 of the Language Reference manual for details on the State Diagram section of the source file.

TEST VECTORS – Contains the optional test vectors that you write to verify the functionality of the logic design. Refer to section 3.9 of the Language Reference manual for additional details. Test vectors can be used in conjunction with PLDtest, which functionally tests the programmed device.

END STATEMENT – Ends the module.

```
module  ....;
flag '........';
title '.........................'

"device declaration
"       name    DEVICE  'device type';
        ...     device  '....';

"pin and node declarations
"       names           PIN  pin #s ;
"       names           NODE node #s ;
        ...,...,..      pin ..,..,..;
        .,.,.,..,.      pin ...,..,..,..,..;
        ...,..,...      node ..,..,..;

"constant declarations
        H,L,X  = 1,0,.X. ;
        ..,..  = ..,..;
        ..     = .. ;

equations
"               name  = expression ;
                 ...  = ... ;
        enable  ...   = ....;
                 ...  := ....;

test_vectors ([...,...]  -> ...)
" inputs                  outputs

 [.,.]  -> [ ..,  ..,  ..,  ..,  ..,  ..,  ..];
 [.,.]  -> [ ..,  ..,  ..,  ..,  ..,  ..,  ..];
 [.,.]  -> [ ..,  ..,  ..,  ..,  ..,  ..,  ..];

end     ....;
```

Figure 3-1. Source File Template

```
module m6809a
title '6809 memory decode
Jean Designer     Data I/O Corp Redmond WA   24 Feb 1984'

                        U09      device  'P14L4';
        A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
        ROM1,IO,ROM2,DRAM       pin 14,15,16,17;

        H,L,X    = 1,0,.X.;
        Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

equations
        !DRAM    = (Address <= ^hDFFF);

        !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);

        !ROM2    = (Address >= ^hF000) & (Address <= ^hF7FF);

        !ROM1    = (Address >= ^hF800);

test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
                ^h0000  -> [  H,  H,  H,   L ];
                ^h4000  -> [  H,  H,  H,   L ];
                ^h8000  -> [  H,  H,  H,   L ];
                ^hC000  -> [  H,  H,  H,   L ];
                ^hE000  -> [  H,  H,  L,   H ];
                ^hE800  -> [  H,  H,  H,   H ];
                ^hF000  -> [  H,  L,  H,   H ];
                ^hF800  -> [  L,  H,  H,   H ];
end m6809a
```

Listing 3-1. Source File Describing an Address Decoder

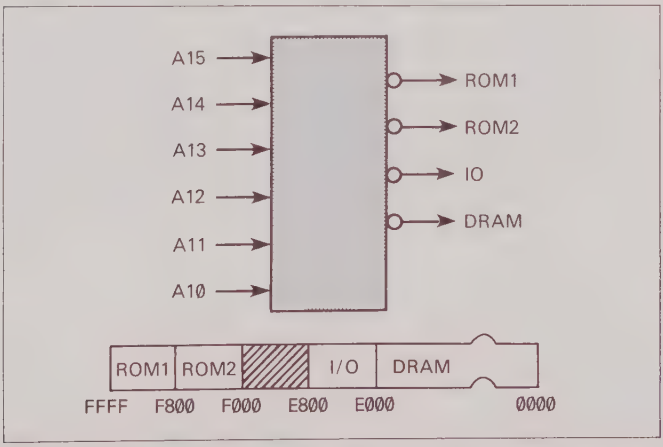


Figure 3-2. Block Diagram: 6809 Memory Address Decoder

EXAMINING AN EXAMPLE ABEL™ SOURCE FILE 3.2

As previously stated, you must write a source file that reflects the design of the circuit to be programmed into the programmable logic device before ABEL™ can generate the required programmer load file. In order to write a meaningful source file, you must be familiar with the make-up and content of an ABEL™ source file.

To familiarize you with the ABEL™ source file, the following paragraphs provide an examination of the source file shown in listing 3-1, which describes the address decoder shown in the block diagram, figure 3-2. This source file is typical of that used with the ABEL™ language processor to produce a programmer load file for downloading to a logic programmer.

The source file shown in listing 3-1 is typical of that used with the language processor and does not illustrate all the features of ABEL™; nor does it show examples of source file syntax rules. For complete source file details, you must refer to the Language Reference and Applications Guide manuals. The Language Reference manual describes the elements and structure of the ABEL™ language; the Applications Guide illustrates many design examples as well as describing special design situations and use of the ABEL™ design simulator. During the following source file examination, reference is made to these manuals.

Purpose of the Address Decoder 3.2.1

An address decoder is a logic design problem that is easily solved with programmable logic. In the address decoder of figure 3-2 the high-order six lines of an address bus are decoded into one of four active-low outputs.

The ROM1 output is low for addresses in the range of F800 to FFFF; the ROM2 output is low for addresses in the range of F000 to F7FF; the I/O output is low for addresses in the range of E000 to EFFF; and the DRAM output is low for addresses in the range of 0000 to DFFF. The outputs will be applied to read-only memory, I/O ports, and dynamic RAM within the target environment to enable these functions when the applicable address appears on the address bus.

The MODULE Statement 3.2.2

The MODULE statement is a required element of the ABEL™ source file. It defines the beginning of the module (since a source file may contain several modules) and must be paired with an END statement. In listing 3-1, the MODULE statement is *module m68O9a*. The keyword is *module*. The identifier *m68O9a* distinguishes the module from any others in the source file. Refer to section 3.2 in the Language Reference manual for a complete description of the MODULE statement.

The FLAG Statement 3.2.3

Although the address decoder example in listing 3-1 does not use the FLAG statement, the source file template in figure 3-1 shows the location of the FLAG statement when used. Refer to section 3.3 in the Language Reference manual for information on the FLAG statement usage.

The TITLE Statement 3.2.4

The TITLE statement may be inserted in the source file to give a title to a module. Although the title is not acted on by the language processor, it will appear as a header in both the programmer load file and documentation file created by the language processor.

The TITLE statement consists of the keyword *title* followed by a string, which is the desired title for the module. The string is opened and closed by an apostrophe. In listing 3-1, the title is:

'6809 memory decode
Jean Designer Data I/O Corp Redmond WA 24 Feb 1984'

Refer to section 3.4 in the Language Reference manual for additional details on the TITLE statement.

The DEVICE Declaration 3.2.5

The DEVICE declaration is used to associate the name of the device with the type of programmable logic device into which the logic design will be programmed. The name of the device in listing 3–1 is *UO9*, which is the schematic reference in this case. The type of programmable logic device is specified as *'P14L4'*. The device name and device type are located on the same line of the source file but separated by *device*, the keyword of the DEVICE declaration. Refer also to section 3.5.1 in the Language Reference manual.

PIN and NODE Declarations 3.2.6

In listing 3–1, the two lines immediately following the DEVICE declaration are the PIN declarations. Two lines are used since there is insufficent horizontal space on an 80–column display to arrange the declaration on a single line. You can use as many lines as necessary to make the pin declarations.

The PIN declaration associates pin identifiers used in the module to actual pin numbers on the target programmable logic device. The first PIN declaration statement associates address line A15 with pin 1 of the P14L4 device, A14 with pin 2, and so on. The second PIN declaration identifies pin 14 of the P14L4 as the ROM1 output, pin 15 as the I/O output, etc.

Attributes can also be assigned to pins, although none are required in this memory decoder example. Refer to section 3.5.2 in the Language Reference manual for complete PIN declaration details.

There are no NODES to be declared in this example since the P14L4 logic device contains no internal signals that are not accessible at the external pins. Refer to section 3.5.3 of the Language Reference maual for information on device nodes.

CONSTANT Declarations 3.2.7

A constant, described fully in section 3.5.4 of the Language Reference manual, is an identifier that retains a constant value through a module. CONSTANT declarations are placed with the PIN and NODE declarations in the source file and use the = sign for their keyword. In listing 3–1, the declaration *H,L,X = 1,0,.X.;* tells the language processor to substitute a logic 1 whenever an upper-case H is encountered, a logic 0 whenever an upper-case L is encountered, and a "don't care" value whenever an upper-case X is encountered. The H, L, and X identifiers are used in subsequent test vectors. (.X. is a special constant used todenote a don't care condition. Refer to section 2.7 of the Language Reference manual.)

The second CONSTANT declaration equates the identifier *Address* with a set consisting of A15, A14, A13, A12, A11, A10, and ten don't care values; i.e., *A15, A14, A13, A12, A11, A10, X,* etc. (The X's are used to account for the low-order address lines, since the "equations" and "test vectors" sections of the source file relate the constant *address* to all 16 address lines.)

EQUATIONS Statements 3.2.8

The EQUATIONS statement defines the beginning of a group of equations that specify the logic functions of a device. The actual equations, written in high-level Boolean equations, follow the EQUATIONS statement. In listing 3-1, there are four equations, one for each of the address decoder outputs. These four equations describe the logic function of the address decoder shown in figure 3-2. Identifiers, defined in the declaration statements, are used in place of P14L4 pin numbers.

The first equation, *!DRAM = (Address = = > hDFFF);* equates the active-low DRAM output (pin 17 of the P14L4) with the address lines set to any address equal to or above DFFF (hex). For the sake of brevity, hexadecimal notation is used to specifiy the address input conditions. ABEL ™ cross-references the identifier *Address* back to the P14L4 pin numbers by means of the preceding CONSTANT and PIN declaration statements.

The second equation equates pin 15 (the active-low I/O output) with the address lines set to any address in the range of E000 to E7FFF. Again, identifiers defined in the declaration statements are used to simplify the equation. The third and fourth equations are written in a similar fashion to define the remaining two address decoder outputs.

Test Vectors 3.2.9

Test vectors specify the expected functional operation of a logic device by defining its outputs as a function of its inputs. (Refer to section 3.9 in the Language Reference manual for more detailed information on test vectors.) Listing 3–1 shows a typical set of test vectors and how they are arranged in a table. The form of the test vector table is determined by the header. In listing 3–1, the test vector header is

test_vectors (Address – > [ROM1,ROM2,IO,DRAM])

The left side of the test vector table specifies the state of the device inputs. Hexadecimal notation is used as a means of simplifying the writing of the vector; all 16 address lines could be indicated individually by 1's and 0's or H's and L's. The right side of the test vector table specifies the state of the device outputs. The H and L identifiers are used to indicate the expected state of each output for the corresponding inputs.

Sufficient test vectors should be included to enable the simulator within ABEL™ to test the intended function of the design. The test vectors provided in listing 3–1 enable testing of the design at several, but not all possible, addresses. The test vectors are sufficient to verify the operation of each output at its lowest address, except for the pin 17 (DRAM) output which is tested at four different addresses.

PROCESSING AN ABEL™ SOURCE FILE 3.3

Once ABEL™ is installed on your system as described in the accompanying Installation and System-Specific Information, you can process a source file to create a programmer load file and, if you wish, download the load file to your logic programmer. This section describes how to process a source file and explains the different processes ABEL™ executes in order to achieve the load file and accompanying documentation. The following procedures use the address decoder source file previously described and shown in listing 3-1.

Entering the Command Line 3.3.1

The ABEL™ command line consists of

ABEL filename [parameters]

ABEL (in the command line) is a batch file (or command script) that contains commands to run all six steps of the language processor automatically (although each step can be executed individually on specified files). The filename (without the extension) specifies the input file, which is the source file to be processed into a programmer load file. The allowable parameters, plus a detailed description of the ABEL batch processing, are presented in section 4 of this manual.

The source file for the address decoder is m6809a.abl and is provided on your Design Examples disk or tape supplied as part of the ABEL™ package. Since this file is provided for you, it is not necessary for you to create this source file. *It is necessary however, for you to copy this file, M68O9A.ABL, from the examples directory to the current directory in preparation for processing by ABEL™.*

To enter the command line, to process the address decoder example, type:

abel m6809a RETURN

(It is not necessary enter the .ABL file extension for the source file when typing the command line.)

If you followed the above procedure, you have started the ABEL™ language processor. Listing 3-2 shows the messages that are displayed at your console. These messages give you processing statistics for each step of the language processor's operation. Processing times vary between systems.

```
PARSE Version 1.00 Copyright (c) 1983,1984 Data I/O Corp.
module m6809a
PARSE complete. Time: 5 seconds

TRANSFOR Version 1.00 Copyright (c) 1983,1984 Data I/O Corp.
module m6809a...
TRANSFORM complete. Time: 8 seconds

REDUCE Version 1.00  Copyright (c) 1983,1984 Data I/O Corp.
module m6809a
_device U09
Simple Reductions
REDUCE complete. Time: 7 seconds

FUSEMAP Version 1.00 Copyright (c) 1983,1984 Data I/O Corp.
module m6809a
_device U09
 6 of 16 terms used
FUSEMAP complete.  Time: 4 seconds

SIMULATE Version 1.00 Copyright (c) 1983,1984 Data I/O Corp.
module m6809a
 8 of 8 vectors in U09 passed
SIMULATE complete. Time: 4 seconds

DOCUMENT Version 1.00  Copyright (c) 1983,1984 Data I/O Corp.
module m6809a
_device U09
DOCUMENT complete.  Time: 4 seconds
```

Listing 3-2. Console Display for the Processing Operation

New Files Created 3.3.2

When the processing is complete, several new files are created. These files are:

filename.ext	Description
M6809A.LST	Listing from PARSE, used to check syntax
U09.JED	Programmer load file for design transfer and input to PLDtest
M6809A.SIM	Simulation output file for error checking
M6809A.DOC	Documentation file for design
M6809A.OUT	Intermediate file for SIMULATE, DOCUMENT, and PLDtest.

The name, *U09*, of the programmer load file is taken from the device identifier in the source file. You may want to look at the listing, plus the simulation (.SIM) and documentation (.DOC) files to see what the language processor creates. Each of these files is discussed fully in section 4.

DOWNLOADING THE PROGRAMMER LOAD FILE 3.4

Downloading the programmer load file differs for each system used to operate ABEL™, and also for the model of logic programmer used. In general, you must set up your system to transmit the *.JED file, via a serial I/O port, to the programmer. The programmer must be set up to receive the load file before the system begins transmission of the load file. The mechanics of establishing the communications link between the system and the programmer depend upon your hardware. In MS-DOS systems, PROMlink provides a convenient way to download programmer load files. Additional information on downloading programmer load files for your particular system is contained in the Installation and System-Specific Information provided with the ABEL™ package.

THE ABEL™ LANGUAGE PROCESSOR 4

The ABEL™ language processor converts logic descriptions to programmer load files that can be downloaded to a logic programmer. The language processor checks your logic description, performs logic reduction, simulates the operation of the programmed device, and creates design documentation.

Processing an ABEL™ source file is a six-step process:

1. PARSE : Reads the source file, checks for correct syntax, expands macros, and acts on directives.
2. TRANSFOR : Converts the description to an intermediate form.
3. REDUCE : Performs logic reduction.
4. FUSEMAP : Creates the programmer load file.
5. SIMULATE : Simulates the function of a programmed device.
6. DOCUMENT: Creates design documentation.

In section 3, you saw how the ABEL command can be used to process a source file. The ABEL command is simply a batch file or command script that invokes all six steps of the language processor. You can either run the batch file to process a design, or you can run each step of the language processor separately. The following subsections discuss both of these choices. Section 4.1 describes the batch file in detail. Sections 4.2 through 4.7 describe the individual steps of the language processor, their options, and how to invoke them. Figure 4–1 shows the processing flow of the language processor.

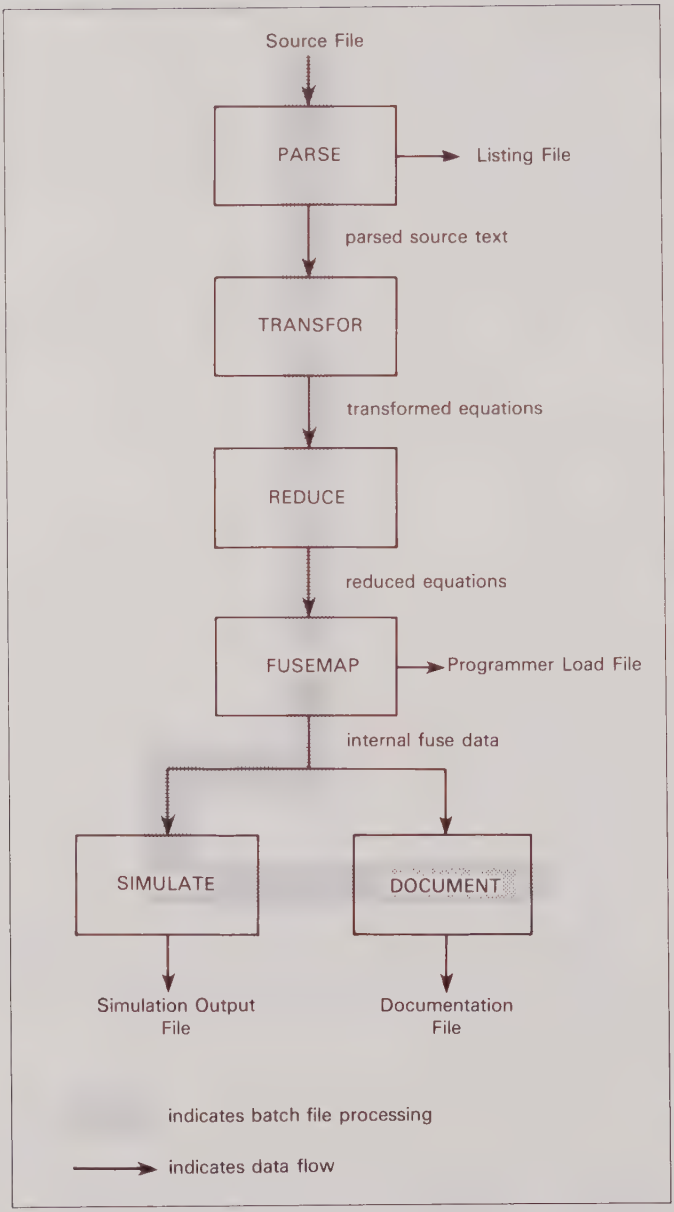


Figure 4-1. Processing Flow of the Language Processor

ABEL BATCH PROCESSING 4.1

ABEL filename [parameters]

filename	the name of an ABEL™ source file. The file extension must be ".ABL" but should not be entered.
parameters	any of the parameters associated with the individual steps of the language processor (except –I, –O, and –B), as described in sections 4.2 through 4.7. Separate parameters by spaces.

ABEL is a batch file that contains the commands needed to run all six steps of the language processor automatically. The input file that you specify with *filename* must be a valid source file. Any devices specified within the source file must be supported by ABEL™, and the corresponding device specification file must be in your library of device specification files. Refer to section 4.1.1, Device File Library.

Any of the parameters associated with the six steps of the language processor, except for –I, –O, and –B, can be entered on the command line to control how a source file is processed. The parameters are discussed in sections 4.2 through 4.7. If you do not specify any parameters, default parameter values take effect and the following actions are performed:

- A parsed listing file is created. Text included with the @INCLUDE directive is shown. Macros, if used, are shown expanded.
- Level 1 logic reduction is performed.
- Programmer load files are created according to the JEDEC standard.
- Level 0 simulation (that shows only simulation errors) is performed.
- Design documentation is created.
- Processing status, statistics and error messages are displayed on the screen.

Device Library 4.1.1

Device specification files are provided with the ABEL™ software package in two forms:

- in a single library file *abellib.dev* that contains specifications on all supported devices
- in individual device files **.DEV*, each of which define a single device. These individual files are those used in earlier versions of ABEL™.

When ABEL is invoked, it will attempt to find a **.DEV* device file that corresponds to each device referenced in the source file. However, if you have included the device library *abellib.dev* in the *abeldev* directory instead of the individual **.DEV* device files, then ABEL™ will then look for the device specifications in the *abellib.dev* device file library.

The device library should be located in the "abeldev" directory on your system. Refer to the Installation and System-Specific Information provided with your ABEL™ package for information on installing the device library.

Refer to section 5.3 for information on the device library manager utility, DEVLIB, used to add devices to and delete devices from *abellib.dev*.

Generated Output Files 4.1.2

Five types of output files are created during execution of ABEL:

filename.LST	parser listing file
device.JED	programmer load files (JEDEC format)
filename.SIM	simulation output
filename.DOC	design documentation
filename.OUT	intermediate file used by SIMULATE and DOCUMENT

CAUTION

During processing, ABEL creates temporary files with the file extensions .TM1, .TM2, and .TM3 and the same filename as the input source file. Any files with a filename and extension identical to those of a temporary file will be overwritten.

Parser listing file. Contains the source code, error messages, and the effect of @INCLUDE directives. (See the ABEL™ Language Reference manual for information on macros and directives.) A complete description of the parser listing file is given in section 4.2.

Programmer load file. Contains the fuse states, test vectors and other design information as defined by the JEDEC standard. This file is loaded into a logic programmer to program and test a programmable logic device. (The programmer load file may also be applied to PLDtest.) One programmer load file is created for each device specified in the source file. Complete specification of the JEDEC standard format is given in appendix B.

Simulation output file. Output from the simulation step indicating whether simulation was completed successfully, and if not, where predicted and actual results differed. See section 4.6 for more information.

Design documentation file. Contains a chip diagram, reduced logic equations and other design information. This file is created by the DOCUMENT step of the language processor.

Intermediate file. The language processor creates a file named filename.OUT that contains intermediate output data created by FUSEMAP. This intermediate file can be used as input to SIMULATE and DOCUMENT (see sections 4.6 and 4.7), as well as PLDtest if available.

Creating Your Own Batch or Command Files 4.1.3

The ABEL batch or command file executes *all* steps of the language processor. You may find that you want to create your own batch file that invokes only some of the steps in the language processor. Or, you may want to run the language processor with special parameter settings at each step. For example, you could write a batch file that runs PARSE, TRANSFOR, REDUCE and FUSEMAP to create a programmer load file without generating design documentation, and without running a simulation.

You can write your own custom batch file by following the rules governing batch files as described in your operating system manual. Combine the individual language processor steps in the way that best suits your application.

PARSE 4.2

```
PARSE [-Iin_file] [-Oout_file] [-Llistfile]
      [-E] [-P] [-Aarg]... [-Hpath] [-W] [-Ypath]
```

-Iin_file	ABEL™ source file is in *in_file*
-Oout_file	intermediate file output from the parser is written to *out_file*. The intermediate file is used as input to TRANSFOR.
-Llistfile	write the listing file to the file specified by *listfile* (standard filename.ext format)
-E	display expanded output resulting from macro expansions and directives as part of the parser listing
-P	display expanded output resulting from macro expansions and directives and show the directive that added code to the source as part of the parser listing
-Aarg	the argument following -A is passed to the PARSE program to be substituted for dummy arguments in the source file
-Hpath	files included in the source with the @INCLUDE directive are located in the directory specified by *path*
-Ypath	any device specification files needed are found in the directory specified by *path*

PARSE reads the source file, converts state diagrams and truth tables to Boolean equations, translates test vectors, expands macros, and checks for correct syntax. If any syntax errors are found, the approximate place at which the error occurs and the type of error are displayed on your monitor. Error messages are also written to a listing file if one is being created (-L). An intermediate file is written to *out_file* if -O is specified. This intermediate file provides input to TRANSFOR.

All parameters are optional. If the –I and/or –O parameters are omitted (no file specified), you will be prompted for the input and/or output file names.

Consider the following example command line:

```
parse -aP14L4 -aGND -im6809a.abl -om6809a.par
-lm6809a.lst -e
```

This command invokes PARSE to process the source file *m68O9a.abl* which contains the logic description for the address decoder described in section 3. The output is written to the file, *m68O9a.par*, and a listing file named *m68O9a.lst* is created. Expanded output is shown in the listing because the –E parameter is included. Two arguments, *P14L4*, and *GND*, are passed to the processor for argument substitution.

Note that the command can be entered in lower–case characters and that the parameters can be specified in any order. Each parameter is discussed in detail below.

–I : Specify Input File

Use –I to specify the file containing your source file. If no input file is specified, the input is assumed to come from your standard input device (usually the keyboard).

–O : Specify Output File

Use –O to specify the name of the output file. If no output file is specified, output will go to your standard output device (usually the monitor). The output file contains the parsed source code and is used as input to the TRANSFOR processor (section 4.3).

–L : Create a Listing File

–L indicates that a listing file is to be created. The file containing the listing is specified directly after the –L parameter. A listing file contains the parsed source code with error messages (if there are any) and with macro expansions and inclusion of code due to directives if the –E parameter was specified. (For information concerning macros and directives, see the Language Reference manual.) If the –P parameter is specified, the listing also shows the directive that caused the inclusion of code. If –L is not used, no listing file will be created.

–E : Show Expanded Output

The –E parameter causes the parsed and expanded source code to be written to the listing file. Text included by macros and directives is shown. (For information concerning macros and directives, see the Language Reference manual.) If –E is not specified, the listing contains the source file as it was before processing plus error messages: expanded text is not shown. If no listing file is specified (the –L parameter is omitted), expanded output will be displayed at your terminal.

–P : Show Expanded Output

In addition to the listing information provided by –E, –P lists the directives that caused code to be added to the source. If the –L parameter is not included to produce a listing file, the directive and the expanded output will be displayed at your terminal.

–A : Pass Arguments to Source

The –A parameter lets you pass arguments to a source file. These arguments are substituted for dummy arguments in the source. As many arguments as are needed can be specified, but each argument must be preceded by the –A parameter. Argument substitution is discussed further in the Language Reference manual.

-H : Specify Path for Included Files

The @INCLUDE directive, described in section 4 of the Language Reference manual, lets you include source text from one file in another source file. By default, the included file is assumed to be in the default directory; -H allows you to override that default value and explicitly specify where included files are to be found. Specify a drive and path immediately after -H. For example, the following specifies that include files are to be found in the directory named *examples*:

```
-H/examples
```

See your operating system manual for more information about path and drive specifications.

-Y : Specify Path for Device Specification Files

The language processor uses device specification files in conjunction with your source file to properly process a design. The -Y parameter lets you specify which drive and directory contains the device files. Specify a path immediately after -Y.

For example,

```
-Y/devices
```

indicates that device specification files are in a system subdirectory, /devices. If -Y is not used, device files are assumed to be in the default directory. See your operating system manual for more information about path and drive specifications.

PARSE Listing File 4.2.1

Listing 4–1 shows a listing file created by the PARSE step of the language processor. This listing was created by running PARSE on an altered version of the M6809A.ABL source file described in section 3. Syntax errors were introduced into the M6809A.ABL source file to create the file M6809ERR.ABL, also provided with the design examples. M6809ERR.ABL contains two syntax errors.

The first error is a missing semicolon in the equation:

```
! DRAM  =  (Address  <  =  ^ hDFFF)
```

The second error is a missing left parenthesis in the truth table header:

```
test_vectors Address -> [ROM1,ROM2,IO,DRAM])
```

These two errors are displayed at your monitor and shown in the listing file. The approximate places at which the error occur are indicated by circumflexes. The type of error is also indicated.

Listing 4–2 shows the corrected source file with the proper semicolon and left parenthesis added.

```
ABEL Version 1.00 Data I/O Corp.

0001 !module m6809err
0002 !title '6809 memory decode
0003 !Jean Designer    Data I/O Corp Redmond WA   24 Feb 1984'
0004 !
0005 !                              U09     device  'P14L4';
0006 !         A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007 !         ROM1,IO,ROM2,DRAM       pin 14,15,16,17;
0008 !
0009 !         H,L,X   = 1,0,.X.;
0010 !         Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];
0011 !
0012 !equations
0013 !         !DRAM   = (Address <= ^hDFFF)
0014 !
0015 !         !IO     = (Address >= ^hE000) & (Address <= ^hE7FF);
                  ^

Syntax error: ';' expected

0016 !
0017 !         !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);
0018 !
0019 !         !ROM1   = (Address >= ^hF800);
0020 !
0021 !test_vectors  Address -> [ROM1,ROM2,IO,DRAM])
                    _       ^

Syntax error: '(' expected

0022 !                  ^h0000  -> [  H,  H,  H,   L ];
0023 !                  ^h4000  -> [  H,  H,  H,   L ];
0024 !                  ^h8000  -> [  H,  H,  H,   L ];
0025 !                  ^hC000  -> [  H,  H,  H,   L ];
0026 !                  ^hE000  -> [  H,  H,  L,   H ];
0027 !                  ^hE800  -> [  H,  H,  H,   H ];
0028 !                  ^hF000  -> [  H,  L,  H,   H ];
0029 !                  ^hF800  -> [  L,  H,  H,   H ];
0030 !end m6809err
```

Listing 4-1. PARSE Listing File with Errors from M6809ERR.ABL

```
module m6809a
title '6809 memory decode
Jean Designer    Data I/O Corp Redmond WA   24 Feb 1984'

                        U09     device  'P14L4';
        A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
        ROM1,IO,ROM2,DRAM       pin 14,15,16,17;

        H,L,X   = 1,0,.X.;
        Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

equations
        !DRAM   = (Address <= ^hDFFF);

        !IO     = (Address >= ^hE000) & (Address <= ^hE7FF);

        !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);

        !ROM1   = (Address >= ^hF800);

test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
                ^h0000  -> [  H,  H,  H,   L ];
                ^h4000  -> [  H,  H,  H,   L ];
                ^h8000  -> [  H,  H,  H,   L ];
                ^hC000  -> [  H,  H,  H,   L ];
                ^hE000  -> [  H,  H,  L,   H ];
                ^hE800  -> [  H,  H,  H,   H ];
                ^hF000  -> [  H,  L,  H,   H ];
                ^hF800  -> [  L,  H,  H,   H ];
end m6809a
```

Listing 4-2. Corrected Source File, M6809A.ABL

TRANSFOR 4.3

TRANSFOR [-Iin_file] [-Oout_file]

-Iin_file	*in_file* is an intermediate output file created by PARSE.
-Oout_file	output from TRANSFOR is written to *out_file*

TRANSFOR reads the equations generated by PARSE and manipulates them to:

1. Replace sets with equivalent equations not involving sets.
2. Replace all operators with equivalent operations using only NOTs, ANDs, ORs and XORs.
3. OR together equations causing multiple assignments to the same identifier.
4. Perform simple logic reduction based on the following rules:

Rule	Description
A & 1 = A	A AND 1 = A
A & 0 = 0	A AND 0 = 0
A # 1 = 1	A OR 1 = 1
A # 0 = A	A OR 0 = A
A $ 1 = !A	A XOR 1 = NOT A
A $ 0 = A	A XOR 0 = A
A !$ 1 = A	A XNOR 1 = A
A !$ 0 = !A	A XNOR 0 = NOT A

The transformed equations can then be reduced by the REDUCE program. The file specifications given for *in_file* and *out_file* must be valid file specifications per your operating system.

Example:

```
TRANSFOR -im6809a.par -om6809a.tsf
```

This command transforms the parsed source code contained in *m6809a.par* (created by PARSE) and creates an output file named *m6809a.tsf*.

REDUCE 4.4

REDUCE [-Iin_file] [-Oout_file] [-Rn]

-Iin_file	*in_file* is an intermediate file created by TRANSFOR.
-Oout_file	output from REDUCE is written to *out_file*
-Rn	n = 0, 1, 2, or 3, indicating the level of reduction to be performed (default = 1): n = 0 : no reduction n = 1 : simple reduction n = 2 : simple and PRESTO[1] reduction n = 3 : simple and PRESTO[1] reduction by pin

REDUCE reduces your logic description so that fewer product terms are used in the programmable logic device. REDUCE reduces the Boolean equations in *in_file* and writes the reduced equations to the file specified by *out_file*. The equations in *in_file* must be in the form produced by TRANSFOR. If *in_file* is not specified, the input is assumed to come from your standard input device (usually your keyboard). If *out_file* is not specified the output is written to your standard output device (usually your monitor).

You may choose one of four levels of reduction: 0, 1, 2, or 3. If you do not explicitly choose a reduction level with -R, level one reduction is performed.

When using exclusive-OR parts, such as P20X8, P20X4, etc., and exclusive-OR conditions are encountered in the input file to REDUCE, REDUCE will leave the exclusive-OR operator intact for that output. If more than one exclusive -OR conditions is encountered for one output, only the first will utilize the exclusive-OR operator; others will be converted to multiple ANDs and ORs.

Level 0 Reduction. No reduction is performed for level 0 reduction. The transformed equations are checked for valid outputs for the specified device and written to the output file.

Level 1 Reduction. If you choose level 1 reduction (–R1), the logic equations are expanded to a sum of products form and then reduced according to the following basic rules of Boolean algebra:

Rule			Description		
! 0	=	1	NOT 0	=	1
!1	=	0	NOT 1	=	0
A & 0	=	0	A AND 0	=	0
A & 1	=	A	A AND 1	=	A
A # 0	=	A	A OR 0	=	A
A # 1	=	1	A OR 1	=	1
A # A	=	A	A OR A	=	A
A & A	=	A	A AND A	=	A
A & !A	=	0	A AND (NOT A)	=	0
A # !A	=	1	A OR (NOT A)	=	1

Level 1 reduction is the default if no level is explicitly chosen with the –R parameter.

Level 2 Reduction. Level 2 reduction should be used whenever the number of product terms used to implement a function is near to or more than the number of product terms available in the device. If you choose level 2 reduction (–r2), both level 1 and PRESTO logic reduction are performed.

The following command line performs simple and PRESTO reduction on the transformed equations contained in *m6809a.tsf* (created by TRANSFOR) and writes the reduced equations to the file named *m6809a.ref*.

```
REDUCE –r2 –im6809a.tsf –om6809a.red
```

Level 3 Reduction. Level 3 reduction reduces logic associated with each pin in a device on a pin-by-pin basis. There are two major advantages to pin reduction. First, reduction by pin is faster than normal PRESTO reduction. Second, this type of reduction is well-suited to PALs which do not share terms among the outputs.

As mentioned, reduce levels 2 and 3 utilize the PRESTO[1] method. PRESTO is a logic reduction technique developed by Antonin Svoboda [1] that reduces the number of inputs and product terms. In the PRESTO method, two functions are formed. The functions are designated F and FDC and are tabular representations of the original equations. During the operation of ABEL™, F and FDC are identical at the beginning of term minimization. PRESTO tries to eliminate each input literal for each product term by temporarily omitting it from F and checking to see if the result is still covered by FDC. If covered, the literal term is deleted from F. After all input literals have been processed, all output literals are similarly processed, only the literal is temporarily omitted from FDC and the checked for coverage by F. If all output literals are eliminated from a given term, the term is removed. The process is repeated until F stabilizes and represents the minimized function.

- Douglas W. Brown, "A State Machine Synthesizer – SMS," **Proceedings of the 18th Design Automation Conference**, IEEE (June, 1981), pp.301–305.

FUSEMAP 4.5

FUSEMAP [–Iin_file] [–Oout_file] [–Jpath] [–Cn] [–Dn] [–Kq]

–Iin_file	*in_file* is an intermediate file created by REDUCE.
–Oout_file	intermediate output from FUSEMAP is written to *out_file*
–Jpath	drive and path specification for programmer load file output
–Cn	checksum parameter. *n* can be:
	0 omit STX, ETX, and transmission checksum from the programmer load file
	1 include STX, ETX, and a dummy transmission checksum in the load file
	2 include full STX, ETX, and transmission checksum (default)
–Dn	programmer load file format specification, where *n* can be either 0, indicating a standard JEDEC format programmer load file, or any of the microprocessor formats defined in table 4–1. JEDEC format is the default.
–Kq	unused fuses in an FPLA OR array will be disconnected or left connected according to the value of *q*:
q = Y	leave unused fuses connected
q = N	disconnect unused fuses (default)

FUSEMAP processes the output of REDUCE and creates programmer load files that are loaded into a logic programmer to program and test devices. One programmer load file is created for each device specified in the original source file. A load file contains fuse states for programming the device and test vectors to test it once it has been programmed. FUSEMAP creates a JEDEC format load file unless you specify a different format with the –D parameter.

The filename of each load file is the name of the device for which the file contains a fuse map. The file extension is "JED" for JEDEC format programmer load files and ".Pxx" for all other load file formats, where *xx* corresponds to the number specified with the –D parameter. The load file is written to the path specified by *Jpath*:, or to the default path if no drive is specified. FUSEMAP also creates an intermediate file specified by *out_file* that is used by the DOCUMENT and SIMULATE steps.

Example:

```
FUSEMAP –im6809a.red –om6809a.fus
```

This command invokes FUSEMAP. The file *m68O9a.red* (output of REDUCE) is read, one programmer load file is created for each device in the source file, and the output file *m68O9a.fus* is created. Assuming that three devices named *DEVICE1*, *DEVICE2*, and *DEVICE3* were declared in the source file, FUSEMAP would create the files:

```
DEVICE1.JED
DEVICE2.JED
DEVICE3.JED
```

Each file is a programmer load file (conforming with the JEDEC standard) that describes the logic and test functions to be programmed into the device. These files can be processed by PLDtest to generate a full set of device test vectors.

-J : Specify drive and path for programmer load file output

The -J parameter allows you to indicate where the programmer load file is written. Specify the desired drive and/or path after -J.

-C : Checksum parameter

Normally, STX is placed at the beginning of the programmer load file and ETX and a checksum are placed at the end of the file in accordance with the JEDEC standard. The -C parameter allows you to control whether and how STX, ETX, and the checksum are written. -C0 omits the STX, ETX, and checksum from the programmer load file. -C1 causes STX and ETX to be written to the programmer load file as usual, but writes a dummy checksum, 0000, thereby disabling checksum checking. -C2 is the default and causes the full STX, ETX, and valid checksum to be written.

-D : Specify Programmer Load File Format

The -D parameter specifies the format of the programmer load file. -D0 is the default and indicates that the programmer load file is to be in the JEDEC format. Other formats can be specified by following -D with the appropriate microprocessor format code (-d83, for example). Supported microprocessor formats, their codes, and the file extension given to the programmer load files are given in table 4-1.

–K : Unused Fuses parameter

Unused fuses in an IFL or FPLA OR array can either be left connected or disconnected. Leaving unused fuses connected allows the addition of logic functions to the device, or alteration of an existing design in the device. However, some speed and power improvements are achieved by disconnecting all unused fuses. Disconnecting the fuses prevents any future modifications to the device.

The –K parameter lets you specify what to do with unused fuses. *–Ky* leaves the fuses connected. *–Kn* disconnects unused fuses. If –K is not specified, the unused fuses are disconnected.

Note: Non–JEDEC programmer load file formats do not contain test vectors, even if test vectors were specified in the source file.

Table 4–1. Data Translation Format Codes and File Extensions

Format	Code	Extension
Motorola Exorciser	82	.P82
Intel Intellec 8/MDS	83	.P83
Motorola Exormax	87	.P87
Intel MCS-86 Hexadecimal Object	88	.P88

SIMULATE 4.6

SIMULATE [-Iin_file] [-Oout_file] [-Tn] [-Ndevice]
[-Bn1,n2[,n3]] [-Xn] [-Zn]

-Iin_file	*infile* is either an intermediate output file created by FUSEMAP or a programmer load file in JEDEC format
-Oout_file	simulation output file
-Tn	*n* is the simulation trace level, where the trace levels are: 0 : show errors only 1 : show test vectors 2 : show output and test vectors 3 : show complete device internals 4 : show waveform default is trace level 0, errors are always shown.
-Ndevice	industry part number for device. Used when simulation is performed independently of ABEL™.
-Bn1,n2[,n3]	set breakpoints *n1* and *n2* between which a new trace level *n3* is in effect. *n1* and *n2* are inclusive.
-Xn	set the logical value used for ".X." in load file test vectors. *n* can be 1,0,H,L. Default is 0.
-Zn	set the logical value used for ".Z." in load file test vectors. *n* can be 1,0,H,L. Default is 1.
-Wn,n,n	produce waveform output for specificed pin numbers. Up to 10 pin numbers may be specified.

If the –I and/or –O parameters are omitted (no file specified), you will be prompted for the input and/or output file names. Also, if the device part number (–Ndevice) is not specified, you will be prompted.

SIMULATE uses design and device information to simulate the operation of a programmable device. SIMULATE can use either the design information created by FUSEMAP or any programmer load file conforming with the JEDEC standard to simulate the operation of PALs, FPLAs, and FPLSs. FUSEMAP intermediate file output must be used to simulate PROMs.

SIMULATE does not execute Boolean equations or apply inputs to ABEL™ truth tables or state diagrams; it simulates the operation of a device as though it were already programmed with the information contained in the input file. If a programmer load file is used as the input to SIMULATE, part number information must be provided by using the –N parameter. If the output file from FUSEMAP (specified with –O at invocation of FUSEMAP) is used for simulation, the part number information is already available and –N is not needed.

Furthermore, if a programmer load file is used as input to SIMULATE, the operation of the one device associated with that load file will be simulated. If the output from FUSEMAP is used as input, the operation of all devices initially specified in the source file will be simulated.

Devices with registered outputs must be clocked before the outputs reflect any change in inputs. The clock pulse must be specified as one of the inputs in the test vectors for such devices. By choosing the appropriate trace level, you can see only the final outputs for registered devices, or the outputs before and after the clock pulse. See section 4.6.2 for an example of simulation output for a registered device.

The outputs of devices with feedback cannot always be determined by one evaluation of the input–to–output function but may require several successive evaluations until the outputs stabilize. SIMULATE uses an iterative method to compute the outputs, evaluating the function repeatedly until the outputs stabilize. If the outputs do not stabilize after 20 iterations, an error is reported. More detailed information on simulating devices with feedback is presented in section 4.6.2.

The output created by SIMULATE is written to the file specified with the –O parameter, or to the standard output device if no output file is given. The contents of this file will vary depending on the trace level set with the –T parameter. See sections 4.6.1 and 4.6.2 for examples of simulation output.

–T : Set Trace Level

–T determines the level of information that is provided by SIMULATE. The trace level can be 0, 1, 2, 3, or 4. Errors are listed regardless of the trace level.

Level 0 shows the final output (after the outputs have stabilized) and test vectors for errors only.

Trace level 1 shows final output and test vectors for the device.

Trace level 2 shows the outputs after each iteration of the simulator and the test vectors. For registered parts, the outputs are shown before and after each clock pulse. For designs with feedback, the outputs are shown for each iteration until the outputs stabilize.

Trace level 3 shows the internal nodes and the device outputs for each iteration in addition to the test vectors. Use level 3 for the most help with determining where and why simulation errors occur.

Trace level 4 shows the output level that appears on each specified device pin during the simulation process. The output pin voltages are shown as a waveform in the output file that contains a trace for each pin. Each trace represents the logic high and logic low output levels for each test vector. The waveforms of up to 10 output pins can be printed from the output file.

Note: PROM designs can only be simulated at trace levels 0, 1, and 4.

-N : Specify a device type to be used for simulation

If a programmer load file is used for simulation input rather than FUSEMAP output, no device-specific information is available. -N specifies the device associated with the load file, and the device information is read from the appropriate device specification file on the distribution disk. The device must be supported by ABEL™, and is specified by an industry part number following -N.

Example:

```
simulate -iU09.jed -om6809a.sim -nP14L4
```

The above example invokes the simulator with the file named *U09.jed* as input, *m6809a.sim* as the output file, and P14L4.DEV as the device specification file. If P14L4.DEV cannot be found, an error message is issued.

Note: -N and JEDEC files cannot be used to simulate PROMs.

-Bn1,n2[,n3] : Set Breakpoints

It may be useful, particularly in large designs, to view simulation output for only some of the test vectors. -B allows this type of selective tracing. A beginning and an ending breakpoint must be specified. A breakpoint is specified with the number of the test vector at which the break is to occur. Beginning with the the first break point (inclusive) a new trace level is set. After the second breakpoint, the trace level is returned to its value before the break.

On some versions of MS-DOS, the use of commas to separate breakpoint values will cause an error. If this occurs, periods may be substituted.

The new trace level can be specified explicitly with *n3*. If the new trace level is not specified, the trace level between breakpoints will be one level higher than before the break occurred.

Example:

```
simulate –im6809a.fus –om6809a.sim –B5,8,3
```

This invokes the simulator with the default trace level 0. At the fifth test vector, the trace level is set to 3 and remains there until after vector 8, when the trace level is reset to level 0.

Example:

```
simulate –t1 –im6809a.fus –oadder.sim –b5,8
```

In this case, the breakpoints are again vectors 5 and 8. But the initial trace level is set to level 1 by the –T parameter and no new trace level is specified with the breakpoints. Thus, the new trace level between vectors 5 and 8 (inclusive) is level 2 (one level higher than 1).

–X and –Z : Set values for "don't care" and "high impedance"

Don't care and high impedance values encountered in test vectors must be given some value during simulation. The –X and –Z parameters allow you to override the default values. As a default, anytime a ".X." is encountered in a test vector the logical value *L* is substituted for it. As a default, *H* is substituted for a ".Z." value. You can specify default values of 0, 1, L, or H for ".X." or ".Z.". The default values are substituted only when ".X." or ".Z." are inputs to a design or outputs that are fed back as inputs. Outputs that are not fed back are shown in simulation output as they exist in the source file, with ".X."s and ".Z."s intact.

The simulator checks the design with a single voltage level for the don't care inputs, while the target circuit may place other levels of the input during actual operations. For complete simulation, it is recommended that you run the SIMULATE operation with the don't-cares, set to 0(flag–XO), and then again with the set to 1 (flag–X1). Refer also to Don't Cares in Simulator located in Section 4 of the Applications Guide.

–W,n,n,n,....: Specify device pins to appear on trace level 4 waveform

When trace level 4 is specified, the –W,n,n,... parameters may be used to specify up to 10 pins of the device. Each pin number is separated by a comma when using the –W parameter. Each specified pin will appear in the output file (refer to 4.6.1) as a trace of a waveform that shows the logic levels appearing at that pin for each of the test vectors during the simulation process. If it is desired to display the waveform for only certain test vectors, use the set breakpoints parameter.

Example:

simulate –t4 –im6809a.out –om6809a.sim –w1, 2,3,4,5,6,17

The above example invokes the simulator with the file named *m68O9a.out* as the input file and *m68O9a.sim* as the output file, with device pins 1 through 6 and 17 specified for the waveform in the output file. Although the device pin numbers are entered in the command line, the listing from the output file indicates the pins by their identifiers named in the pin declarations of the ABEL™ source file.

SIMULATE Output File 4.6.1

Figure 4–2 shows the output of SIMULATE created during processing of M6809A.ABL (listing 4–2), with pointers to the various parts of the output. For the purposes of this description, one of the test vectors was changed to produce an error, and simulation was run at trace level 0 (only errors are shown). Figure 4–3 shows the test vectors used to create the simulation error, in which the fourth vector has been changed. If the original test vectors shown in listing 4–2 were used, no simulation errors would occur.

The simulation output shows the actual signal outputs and the test vectors used to perform the simulation. The actual output associated with each test vector is shown on one line followed by the test vector on the next line. The test vector lists the inputs on the left side of the symbol, "->", and the expected results on the right. The actual results are also shown to the right for easy comparison. An *N* occurring in the test vectors means that that particular result is not tested. Periods, ".", are place holders for pins that are not outputs.

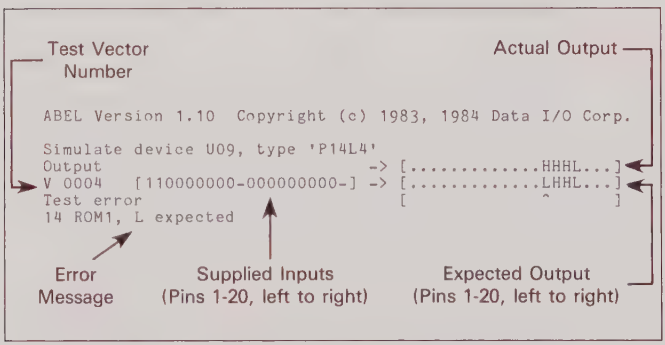

Test Vector Number
Actual Output
ABEL Version 1.10 Copyright (c) 1983, 1984 Data I/O Corp.
Simulate device U09, type 'P14L4'
Output -> [.....HHHL...]
V 0004 [110000000-000000000-] -> [.....LHHL...]
Test error [^]
14 ROM1, L expected
Error Message
Supplied Inputs (Pins 1-20, left to right)
Expected Output (Pins 1-20, left to right)

Figure 4–2. Simulation Output for M6809A.ABL

Listing 4-3 shows a trace level 1 simulation listing for the same source file (including the incorrect vector). Because this design has neither feedback loops nor registered outputs, level 2 simulation provides no extra information and is not shown here.

```
ABEL Version 1.10  Data I/O Corp.

Simulate device U09, type 'P14L4'

Vector 1
Output                                -> [.............HHHL...]
V 0001   [000000000-000000000-] -> [.............HHHL...]

Vector 2
Output                                -> [.............HHHL...]
V 0002   [010000000-000000000-] -> [.............HHHL...]

Vector 3
Output                                -> [.............HHHL...]
V 0003   [100000000-000000000-] -> [.............HHHL...]

Vector 4
Output                                -> [.............HHHL...]
V 0004   [110000000-000000000-] -> [.............LHHL...]
Test error                            [                ^      ]
14 ROM1, L expected


Vector 5
Output                                -> [.............HLHH...]
V 0005   [111000000-000000000-] -> [.............HLHH...]

Vector 6
Output                                -> [.............HHHH...]
V 0006   [111010000-000000000-] -> [.............HHHH...]

Vector 7
Output                                -> [.............HHLH...]
V 0007   [111100000-000000000-] -> [.............HHLH...]

Vector 8
Output                                -> [.............LHHH...]
V 0008   [111110000-000000000-] -> [.............LHHH...]
```

Listing 4-3. Trace Level 1 Simulation Output

```
test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
              ^h0000  -> [  H,   H,   H,    L ];
              ^h4000  -> [  H,   H,   H,    L ];
              ^h8000  -> [  H,   H,   H,    L ];
              ^hC000  -> [  L,   H,   H,    L ];
              ^hE000  -> [  H,   H,   L,    H ];
              ^hE800  -> [  H,   H,   H,    H ];
              ^hF000  -> [  H,   L,   H,    H ];
              ^hF800  -> [  L,   H,   H,    H ];
```

Figure 4-3. Test Vectors Used to Create Simulation Errors

Figure 4-4 shows one section of a trace level 3 listing with pointers to its various parts and a section of a PAL logic diagram. Refer to figure 4-4 as you read the following description. Trace level 3 simulation shows a pin table, OR-gate outputs, register outputs (if any exist) and the final outputs. The pin table lists the pin identifiers and their associated pin numbers, fuse numbers and the values of the product terms during simulation. Fuse numbers shown on the table are numbers assigned by Data I/O to the fuses in the device and are shown in the Programmable Device Logic Diagrams manual, document number 981-0017. The OR-gate and register outputs shown in the simulation output are internal signals not available as pin outputs that can be very useful for debugging designs.

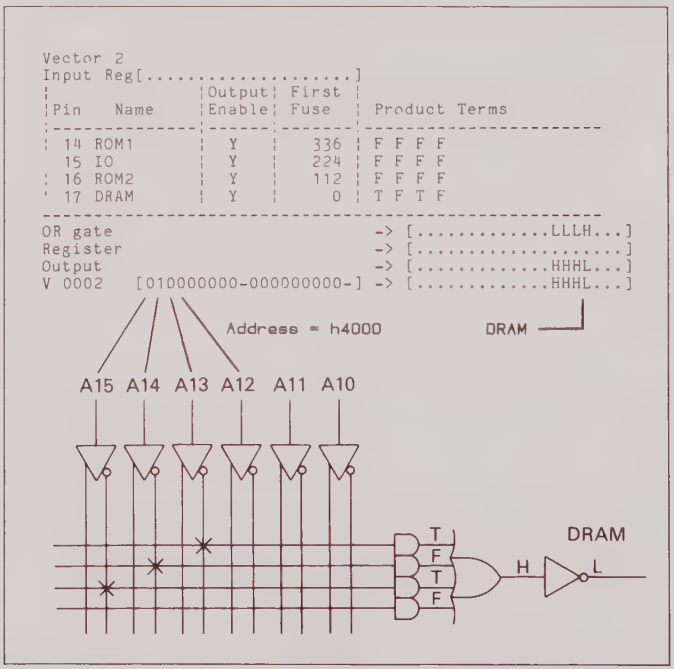


Figure 4-4. Trace Level 3 Simulation Output

Figure 4–5 shows the waveform given of a trace level 4 listing of pins 1 through 6 and 17 of the address decoder described in section 3 of this manual. To obtain this listing, SIMULATE was run from the file *m68O9a.out* which is the intermediate file of the address decoder generated by ABEL™. Each of the specified pins is shown with the logic high and low levels for each of the eight test vectors. The device type and name given at the top of the waveform are taken from the intermediate file specified in the SIMULATE command line.

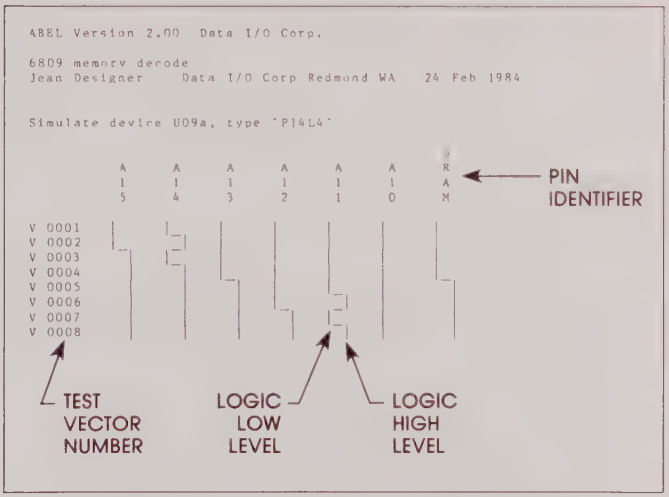


Figure 4–5. Trace Level 4 Simulation Output

Simulation for Designs With Feedback 4.6.2

Logic designs containing feedback present a unique simulation problem because the current output on one or more gates in the design depends on the outputs of other gates. Thus, determining the outputs of a design with feedback is not a simple input-to-output determination. Propagation delays, the number of gates in the feedback path, and, in synchronous feedback circuits, clocks must be taken into account. When an input to the design changes, the outputs may not stabilize immediately. Synchronous circuits must be clocked before the outputs reflect changes in the inputs.

SIMULATE determines the final outputs of feedback circuits through iteration, calculating and monitoring the outputs until they stabilize or are clocked out to give the final outputs. (If outputs do not stabilize after 20 iterations, an error message is given.) The iterations, final outputs and the states of the internal register are provided in the simulation output file depending on the trace level you choose to simulate under. Figure 4-6 shows a simple synchronous circuit with feedback. One clock pulse is required after the inputs change to cause a corresponding change in the outputs. The source file describing this circuit and the simulation output for trace levels 1-3 are shown in listings 4-4 through 4-7. Trace level 0 output lists the device type and name, and simulation errors (none exist for this design) and is not shown.

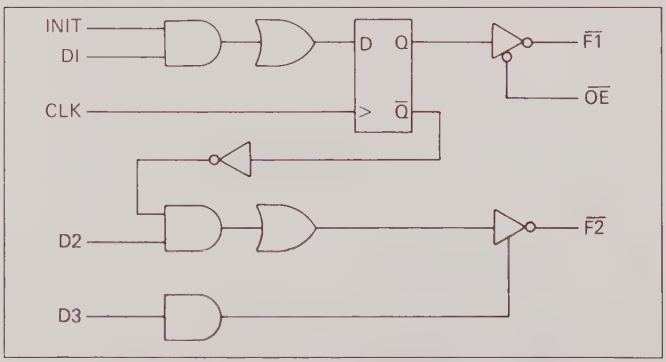


Figure 4-6. Synchronous Feedback Circuit

Trace level 1 output shows the test vectors and the final outputs after the clock pulse. Trace level 2 shows the test vectors and the value of the outputs before and after the clock. The clock pulse is indicated by *## Clock ##* in the listing. Trace level 3 shows a pin table, the OR gate outputs, the register outputs and the final outputs as they exist before and after the clock.

```
module regfb
title 'Operation of the simulator on devices with feedback
       DATA I/O Corp.      24 Feb 1984'

        FB1     device 'P16R4';

        Clk,OE          pin 1,11;
        INIT,D1,D2,D3   pin 2,3,4,5,;
        F1,F2           pin 14,13;

equations
        !F1     := D1 & INIT;
        !F2      = D2 & !F1;

        ENABLE  F2 = D3;

test_vectors    ([Clk,OE,INIT,D1,D2,D3] -> [ F1, F2])
                 [.C., 0,  0 , 1, 1, 1] -> [ 1,  1 ];
                 [.C., 1,  0 , 0, 0, 0] -> [.Z.,.Z.];
                 [.C., 1,  1 , 1, 1, 1] -> [.Z., 0 ];
                 [ 0 , 0,  0 , 0, 0, 0] -> [ 0 ,.Z.];
end regfb
```

Listing 4–4. Source File: Synchronous Feedback Circuit

```
ABEL Version 1.10  Data I/O Corp.

Simulate device FB1, type 'P16R4'

Vector 1
Output                                  -> [...........ZHHHHHZZ.]
V 0001   [C01110000-000000000-]  -> [...........NHHNNNNN.]

Vector 2
Output                                  -> [...........ZZZZZZZZ.]
V 0002   [C00000000-100000000-]  -> [...........NZZNNNNN.]

Vector 3
Output                                  -> [...........ZLZZZZZZ.]
V 0003   [C11110000-100000000-]  -> [...........NLZNNNNN.]

Vector 4
Output                                  -> [...........ZZLHHHZZ.]
V 0004   [000000000-000000000-]  -> [...........NZLNNNNN.]
```

Listing 4–5. Simulation Output, Trace Level 1: Synchronous Feedback Circuit

```
ABEL Version 1.10  Data I/O Corp.

Simulate device FB1, type 'P16R4'

Vector 1
Output                              -> [..........ZHHHHHZZ.]
Clock    [.....................]      [............CCCC...]
V 0001   [C01110000-000000000-] -> [..........NHHNNNNN.]

Vector 2
Output                              -> [..........ZZZZZZZZ.]
Clock    [.....................]      [............CCCC...]
V 0002   [C00000000-100000000-] -> [..........NZZNNNNN.]

Vector 3
Output                              -> [..........ZHZZZZZZ.]
Clock    [.....................]      [............CCCC...]
Output                              -> [..........ZLZZZZZZ.]
V 0003   [C11110000-100000000-] -> [..........NLZNNNNN.]

Vector 4
Output                              -> [..........ZZLHHHZZ.]
V 0004   [000000000-000000000-] -> [..........NZLNNNNN.]
```

Listing 4–6. Simulation Output, Trace Level 2: Synchronous Feedback Circuit

```
ABEL Version 1.10  Data I/O Corp.

------------------------------------ FB1 -----------------
type 'P16R4'

Vector 1
Input Reg[.....................]
|               |Output| First |
|Pin    Name    |Enable| Fuse  |  Product Terms
|---------------|------|-------|-----------------------
| 12            |   N  |  1824 | F F F F F F F
| 13 F2         |   Y  |  1568 | F F F F F F F
| 14 F1         |   Y  |  1280 | F F F F F F F F
| 15            |   Y  |  1024 | F F F F F F F F
| 16            |   Y  |   768 | F F F F F F F F
| 17            |   Y  |   512 | F F F F F F F F
| 18            |   N  |   288 | F F F F F F F
| 19            |   N  |    32 | F F F F F F F
-------------------------------------------------------
OR gate                             -> [..........LLLLLLLL.]
Register                            -> [............LLLL...]
Output                              -> [..........ZHHHHHZZ.]
Clock    [.....................]      [............CCCC...]
V 0001   [C01110000-000000000-] -> [..........NHHNNNNN.]

Vector 2
Input Reg[.....................]
|               |Output| First |
|Pin    Name    |Enable| Fuse  |  Product Terms
|---------------|------|-------|-----------------------
| 12            |   N  |  1824 | F F F F F F F
| 13 F2         |   N  |  1568 | F F F F F F F
| 14 F1         |   N  |  1280 | F F F F F F F F
| 15            |   N  |  1024 | F F F F F F F F
| 16            |   N  |   768 | F F F F F F F F
| 17            |   N  |   512 | F F F F F F F F
| 18            |   N  |   288 | F F F F F F F
| 19            |   N  |    32 | F F F F F F F
-------------------------------------------------------
OR gate                             -> [..........LLLLLLLL.]
Register                            -> [............LLLL...]
Output                              -> [..........ZZZZZZZZ.]
Clock    [.....................]      [............CCCC...]
V 0002   [C00000000-100000000-] -> [..........NZZNNNNN.]
```

Listing 4–7. Simulation Output, Trace Level 3: Synchronous Feedback Circuit

```
Vector 3
Input Reg[.....................]
|                 |Output| First |
|Pin    Name      |Enable| Fuse  |  Product Terms
|-----------------|------|-------|-----------------------
| 12              |   N  |  1824 | F F F F F F F
| 13 F2           |   Y  |  1568 | F F F F F F F
| 14 F1           |   N  |  1280 | T F F F F F F F
| 15              |   N  |  1024 | F F F F F F F F
| 16              |   N  |   768 | F F F F F F F F
| 17              |   N  |   512 | F F F F F F F F
| 18              |   N  |   288 | F F F F F F F
| 19              |   N  |    32 | F F F F F F F
-------------------------------------------------------------
OR gate                              -> [...........LLHLLLLL.]
Register                             -> [.............LLLL...]
Output                               -> [...........ZHZZZZZZ.]
Clock     [.....................]        [.............CCCC...]
Input Reg[.....................]
|                 |Output| First |
|Pin    Name      |Enable| Fuse  |  Product Terms
|-----------------|------|-------|-----------------------
| 12              |   N  |  1824 | F F F F F F F
| 13 F2           |   Y  |  1568 | T F F F F F F
| 14 F1           |   N  |  1280 | T F F F F F F F
| 15              |   N  |  1024 | F F F F F F F F
| 16              |   N  |   768 | F F F F F F F F
| 17              |   N  |   512 | F F F F F F F F
| 18              |   N  |   288 | F F F F F F F
| 19              |   N  |    32 | F F F F F F F
-------------------------------------------------------------
OR gate                              -> [...........LHHLLLLL.]
Register                             -> [.............HLLL...]
Output                               -> [...........ZLZZZZZZ.]
V 0003   [C11110000-100000000-] -> [...........NLZNNNNN.]

Vector 4
Input Reg[.....................]
|                 |Output| First |
|Pin    Name      |Enable| Fuse  |  Product Terms
|-----------------|------|-------|-----------------------
| 12              |   N  |  1824 | F F F F F F F
| 13 F2           |   N  |  1568 | F F F F F F F
| 14 F1           |   Y  |  1280 | F F F F F F F F
| 15              |   Y  |  1024 | F F F F F F F F
| 16              |   Y  |   768 | F F F F F F F F
| 17              |   Y  |   512 | F F F F F F F F
| 18              |   N  |   288 | F F F F F F F
| 19              |   N  |    32 | F F F F F F F
-------------------------------------------------------------
OR gate                              -> [...........LLLLLLLL.]
Register                             -> [.............HLLL...]
Output                               -> [...........ZZLHHHZZ.]
V 0004   [000000000-000000000-] -> [...........NZLNNNNN.]
```

Listing 4–7. *(Continued)*

As a second example, figure 4–7 shows an asynchronous circuit that requires more than one simulation iteration before the outputs stabilize. Listing 4–8 shows the source file describing the circuit and listings 4–9 through 4–11 shows the simulation output for trace levels 1 through 3. Trace level 0 output is not shown because no simulation errors exist for this design.

Trace level 1 shows the final outputs after they have stabilized, and the test vectors. Trace level 2 shows the output values at the different iterations as the outputs stabilize, as well as the final outputs and the test vectors. Notice that for the inputs provided in vector 2, three iterations are needed before the outputs stabilize. Vector 1 requires only one iteration to provide stable outputs. Trace level 3 output contains the pin table, OR gate outputs, register outputs, and final outputs for each iteration until the outputs stabilize.

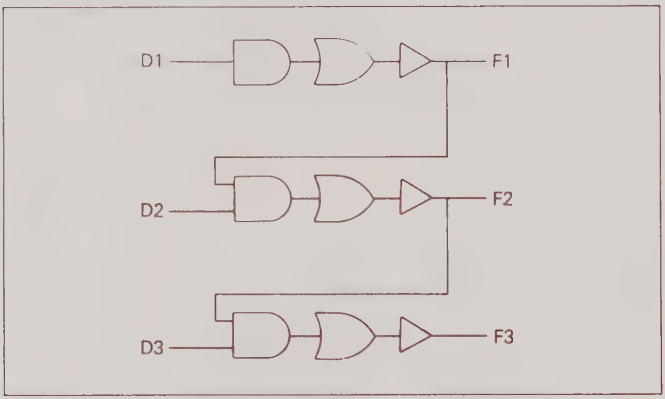


Figure 4–7. Asynchronous Feedback Circuit

```
module feedback
title
'Operation of the simulator on devices with feedback
                    DATA I/O Corp.      24 Feb 1983'

        FB1     device 'P16HD8';

        D1,D2,D3        pin 1,2,3;
        F1,F2,F3        pin 13,14,15;

equations

        F1      = D1;
        F2      = D2 & F1;
        F3      = D3 & F2;

test_vectors    ([D1,D2,D3] -> [F1,F2,F3])
                 [ 0, 0, 0] -> [ 0, 0, 0];
                 [ 1, 1, 1] -> [ 1, 1, 1];
end feedback
```

Listing 4–8. Source File: Asynchronous Feedback Circuit

```
ABEL Version 1.10  Data I/O Corp.

Simulate device FB1, type 'P16HD8'

Vector 1
Output                              -> [...........LLLLLLLL.]
V 0001   [000000000-000000000-] -> [...........NLLLNNNN.]

Vector 2
Output                              -> [...........LHHHLLLL.
V 0002   [111000000-000000000-] -> [...........NHHHNNNN.
```

Listing 4-9. Simulation Output, Trace Level 1: Asynchronous Feedback Circuit

```
ABEL Version 1.10  Data I/O Corp.

Simulate device FB1, type 'P16HD8'

Vector 1
Output                              -> [...........LLLLLLLL.]
V 0001   [000000000-000000000-] -> [...........NLLLNNNN.]

Vector 2
Output                              -> [...........LHLLLLL...
Output                              -> [...........LHHL.....]
Output                              -> [...........LHHHLLLL.]
V 0002   [111000000-000000000-] -> [...........NHHHNNNN.
```

Listing 4-10. Simulation Output, Trace Level 2: Asynchronous Feedback Circuit

```
ABEL Version 1.10  Data I/O Corp.

------------------------------------ FB1 ----------------
type 'P16HD8'

Vector 1
Input Reg[.....................]
                    |Output| First |
Pin    Name         |Enable| Fuse  | Product Terms
--------------------|------|-------|-----------------------
 12                 |   Y  |  1792 | F F F F F F F F
 13 F1              |   Y  |  1536 | F F F F F F F F
 14 F2              |   Y  |  1280 | F F F F F F F F
 15 F3              |   Y  |  1024 | F F F F F F F F
 16                 |   Y  |   768 | F F F F F F F F
 17                 |   Y  |   512 | F F F F F F F F
 18                 |   Y  |   256 | F F F F F F F F
 19                 |   Y  |     0 | F F F F F F F F
-------------------------------------------------------------
OR gate                                -> [...........LLLLLLLL.]
Register                               -> [.....................]
Output                                 -> [...........LLLLLLLL.]
V 0001   [000000000-000000000-] -> [...........NLLLNNNN.]

Vector 2
Input Reg[.....................]
                    |Output| First |
Pin    Name         |Enable| Fuse  | Product Terms
--------------------|------|-------|-----------------------
 12                 |   Y  |  1792 | F F F F F F F F
 13 F1              |   Y  |  1536 | T F F F F F F F
 14 F2              |   Y  |  1280 | F F F F F F F F
 15 F3              |   Y  |  1024 | F F F F F F F F
 16                 |   Y  |   768 | F F F F F F F F
 17                 |   Y  |   512 | F F F F F F F F
 18                 |   Y  |   256 | F F F F F F F F
 19                 |   Y  |     0 | F F F F F F F F
-------------------------------------------------------------
OR gate                                -> [...........LHLLLLLL.]
Register                               -> [.....................]
Output                                 -> [...........LHLLLLLL.]
Input Reg[.....................]
                    |Output| First |
Pin    Name         |Enable| Fuse  | Product Terms
--------------------|------|-------|-----------------------
 12                 |   Y  |  1792 | F F F F F F F F
 13 F1              |   Y  |  1536 | T F F F F F F F
 14 F2              |   Y  |  1280 | T F F F F F F F
 15 F3              |   Y  |  1024 | F F F F F F F F
 16                 |   Y  |   768 | F F F F F F F F
 17                 |   Y  |   512 | F F F F F F F F
 18                 |   Y  |   256 | F F F F F F F F
 19                 |   Y  |     0 | F F F F F F F F
-------------------------------------------------------------
OR gate                                -> [...........LHHLLLLL.]
Register                               -> [.....................]
Output                                 -> [...........LHHLLLLL.]
Input Reg[.....................]
                    |Output| First |
Pin    Name         |Enable| Fuse  | Product Terms
--------------------|------|-------|-----------------------
 12                 |   Y  |  1792 | F F F F F F F F
 13 F1              |   Y  |  1536 | T F F F F F F F
 14 F2              |   Y  |  1280 | T F F F F F F F
 15 F3              |   Y  |  1024 | T F F F F F F F
 16                 |   Y  |   768 | F F F F F F F F
 17                 |   Y  |   512 | F F F F F F F F
 18                 |   Y  |   256 | F F F F F F F F
 19                 |   Y  |     0 | F F F F F F F F
-------------------------------------------------------------
OR gate                                -> [...........LHHHLLLL.]
Register                               -> [.....................]
Output                                 -> [...........LHHHLLLL.]
V 0002   [111000000-000000000-] -> [...........NHHHNNNN.]
```

Listing 4-11. Simulation Output, Trace Level 3: Asynchronous Feedback Circuit

DOCUMENT 4.7

DOCUMENT [-Iin_file] [-Oout_file] [-V] [-Fn] [-G] [-S] [-Qxyz]...

-Iin_file	*infile* is an intermediate output file from FUSEMAP
-Oout_file	design documentation file
-V	list the test vectors
-Fn	list the fuse map and/or terms n = 0 list the fuse map and the device utilization information n = 1 list only the device utilization information
-G	list the chip diagram
-S	list the symbol table
-Qxyz	select equations to be listed, where *x*, *y*, and *z* can be any combination of the following: 2 : list reduced equations 1 : list transformed equations 0 : list original equations

DOCUMENT creates design documentation from information provided by previous steps of the language processor. The documentation is written to the file specified by the -O parameter or to the standard output device in the event that no output file is specified. If the -I and/or -O parameters are omitted (no file specified), you will be prompted for the input and/or output file names.

The design documentation contains the following information for each device in the source file if the appropriate parameter is set:

Symbol table	constant, pin, node, module and macro identifiers listed alphabetically.
Reduced equations	equations produced by REDUCE.
Transformed equations	produced by equations TRANSFOR.
Equations	original equations from the source file and equations generated by PARSE from truth tables and state diagrams.
Test vectors	test vectors described as inputs and outputs, taken from the PARSE intermediate output file.
Fuse map	graphical representation of the fuse states from the programmer load file.
Chip Diagram	a diagram showing the device pinouts and the identifiers assigned to each pin.

If no parameters are supplied in the DOCUMENT invocation, no documentation listing will be generated.

-Q : List Equations Parameter

The -Q parameter controls which equations, if any, appear in the documentation output file. Up to three numbers (0, 1, or 2) can be specified following -Q, or the -Q flag can be repeated with a different number for each type of equations listing desired. For example, both "-q02" and "-q0 -q2" cause the original and reduced equations to be included as part of the documentation output file.

-F : List Fuse Map and Terms Utilization

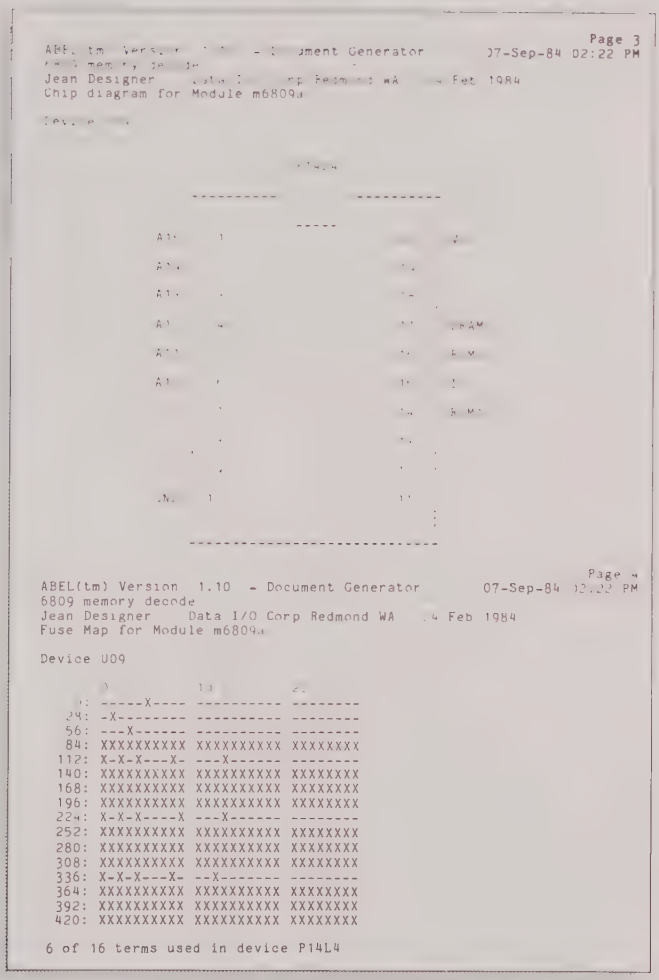
The -F parameter controls the listing of fuse maps and terms. -F0 or -F lists a complete fuse map, the number of terms used, and the utilization of each term with respect to the outputs (utilization report). -F1 lists only the number of terms used and the utilization report.

DOCUMENT Output File 4.7.1

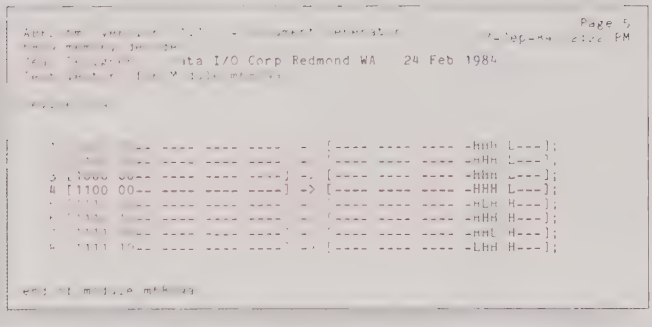
Listing 4-12 shows output from DOCUMENT. This output was created by processing the source file, M6809A.ABL (listing 4-2) using the parameters -V, -F, -G, -S, and -Q2. The listing contains a symbol table, the reduced equations, a chip diagram, a fuse map, and the test vectors. In the fuse map, intact connections are shown as "X"s, and blown fuses (no connection) are shown as dashes. The test vector inputs are shown on the left side of the "->" symbol; outputs appear on the right side.

```
                                                                    Page 1
ABEL(tm) Version  1.10  - Document Generator          07-Sep-84 02:22 PM
6809 memory decode
Jean Designer     Data I/O Corp Redmond WA    24 Feb 1984
Symbol list for Module m6809a


A10                             Pin 6  pos, com
A11                             Pin 5  pos, com
A12                             Pin 4  pos, com
A13                             Pin 3  pos, com
A14                             Pin 2  pos, com
A15                             Pin 1  pos, com
Address                         ([A15,A14,A13,A12,A11,A10,.X.,.X.,.X.,
                                    .X.,.X.,.X.,.X.,.X.,.X.,.X.,.X.])
DRAM                            Pin 17  neg, com
H                               (1)
IO                              Pin 15  neg, com
L                               (0)
ROM1                            Pin 14  neg, com
ROM2                            Pin 16  neg, com
U09                             device P14L4
X                               (.X.)
m6809a                          Module Name


                                                                    Page 2
ABEL(tm) Version  1.10  - Document Generator          07-Sep-84 02:22 PM
6809 memory decode
Jean Designer     Data I/O Corp Redmond WA    24 Feb 1984
Equations for Module m6809a

Device U09


     Reduced Equations:

       DRAM = !(!A13 # !A14 # !A15);

       IO = !(!A11 & !A12 & A13 & A14 & A15);

       ROM2 = !(!A11 & A12 & A13 & A14 & A15);

       ROM1 = !(A11 & A12 & A13 & A14 & A15);
```

Listing 4-12. Documentation Output for M6809A.ABL



```
                                                                    Page 3
ABEL(tm) Version 1.10 - Document Generator          07-Sep-84 02:22 PM
6809 memory decode
Jean Designer     Data I/O Corp Redmond WA    4 Feb 1984
Chip diagram for Module m6809a

Device U09

                                    ---------
                                    |       |
                                    |       |
                                    |       |
```



Listing 4–12. *(Continued)*

5 PROGRAMMER LOAD FILE TRANSFER

When you have completed the processing of an ABEL™ source file, a file named *filename.JED* will exist in your working directory. This file is the programmer load file, and contains the data necessary for a logic programmer to program a logic device with your design. Transferring the programmer load file amounts to downloading it from your system to the logic programmer over an RS232-compatible communications link. Data I/O's PROMlink 2.0 can be used to transfer programmer load files from MS-DOS systems. IF PROMlink is used, refer to the PROMlink manual for details.

DOWNLOADING TO A MODEL 29 WITH LOGICPAK™) 5.1

The following steps are necessary to transfer a programmer load file from an IBM PC/XT/AT (or compatible) to a Data I/O Model 29 programmer with a LogicPak™ installed. *If you are operating with a system other than an IBM PC/XY/AT or compatible, refer to the Installation and System Specific Information provided with your ABEL™ package for variations in the downloading procedure.*

1. Connect the cable (see figures 5-1 and 5-2) to the COM1: port on the IBM PC and to the serial interface of the Model 29.

2. Configure the Model 29 for 4800 baud with no parity (see the Model 29 manual). Parity should be set with the power off.

3. Configure the IBM PC for 4800 baud (9600 baud could also be used) with no parity by issuing the following command at the DOS prompt on the PC:

   MODE COM1: 4800,n,8 RETURN

4. Enter the family code and pinout code of the device to be programmed into the Model 29 as follows:

VERIFY	RAM	DEVICE	START	XXXX

   where *XXXX* is the 4-digit family and pinout code. A list of these codes may be found in the LogicPak™ manual.

5. Prepare the Model 29 to receive the load file. Enter the following at the 29:

SELECT	E	B	START

   The Model 29 will display a stationary action symbol.

6. At the PC, execute the command:

   COPY d:filename.ext COM1: RETURN

where *d* is the drive specification, and *filename* and *ext* are the filename and extension of the programmer load file. The load file extension is ".JED". The Model 29 action symbol should rotate when the file is being transferred.

7. When the transfer is complete, the Model 29 will display the fuse checksum signifying that the programmer load file is now resident in programmer memory.

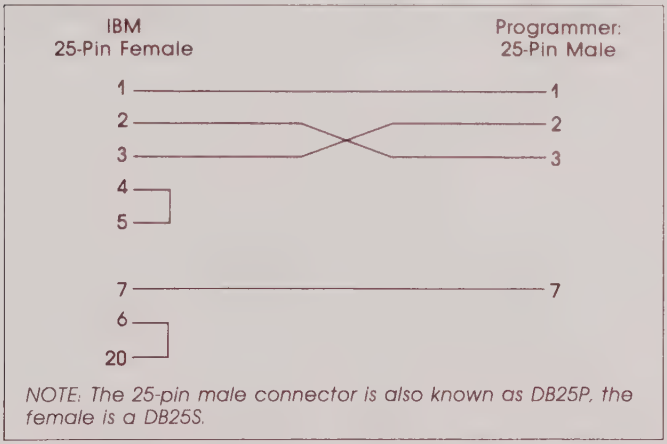


Figure 5–1. Cable Configuration for Transfer Between an IBM PC/XT and a Data I/O Programmer

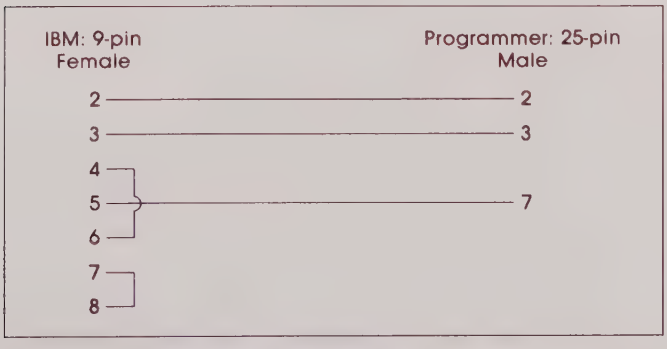


Figure 5–2. Cable Configuration for Transfer Between an IBM-AT and a Data I/O Programmer

DOWNLOADING TO A MODEL 60 PROGRAMMER 5.2

The following steps are necessary to transfer a programmer load file from an IBM PC/XT/AT (or compatible) to a Data I/O Model 60 Logic Programmer. If you are operating with a system other than an IBM PC/XT/AT or compatible, refer to the System Specific Information provided with your ABEL™ package for variations in the downloading procedure.

Data I/O's PROMlink 2.0 can be used to transfer programmer load files from MS-DOS systems. If PROMlink is used, refer to the PROMlink manual for details.

1. Connect the cable (see figure 5-1) to the COM1: port on the IBM PC and to the serial interface of the Model 60.

2. Configure the Model 60 for 4800 baud with no parity (see the Model 60 manual).

3. Configure the IBM PC for 4800 baud with no parity by issuing the following command at the DOS prompt on the PC:

   MODE COM1: 4800,n,8 RETURN

4. Prepare the Model 60 to receive the load file. Enter the following at the 60:

INPUT	START

   The Model 60 will display a stationary action symbol.

5. At the PC, execute the command:
   COPY d:filename.ext COM1: RETURN

   where *d* is the drive specification, and *filename* and *ext* are the filename and extension of the programmer load file. The load file extension is ".JED". The Model 60 action symbol should rotate when the file is being transferred.

6. When the transfer is complete, the Model 60 will display the fuse checksum signifying that the programmer load file is now resident in programmer memory.

PROM DOWNLOAD (MODEL 29 WITH UNIPAK 2™) 5.3

Unless you are using PROMlink, the following steps are necessary to transfer a programmer load file from the IBM PC to a DATA I/O Model 29 programmer with a UniPak 2™:

1. Connect the cable (see figure 5-1) to the COM1: port on the IBM PC and to the serial interface of the Model 29.

2. Configure the Model 29 for 4800 baud with no parity (see the Model 29 manual. Parity should be set with the power off.

3. Configure the IBM PC for 4800 baud with no parity by issuing the following command at the DOS prompt on the PC:

   MODE COM1: 4800,n,8 RETURN

4. Select the microprocessor transfer format of the load file. A list of supported microprocessor formats may be found in in table 4-1. For example, the code for the Motorola Exorciser format is 82. To select this format enter the following at the 29:

SELECT	8	2	START	START

5. Prepare the Model 29 to receive the load file. Enter the following at the 29:

COPY	PORT	RAM	START

   The Model 29 will display a stationary action symbol.

6. At the PC, execute the command:

   COPY d:filename.ext COM1:

   where *d* is the drive specification, and *filename* and *ext* are the filename and extension of the programmer load file. The load file extension for PROMs is ".Pxx". where xx is the format code (see table 4–1). The Model 29 action symbol should rotate when the file is being transferred.

7. When the transfer is complete, the Model 29 will display the the fuse RAM sumcheck.

This section describes the utilities provided in addition to the ABEL™ software.

TOABEL, PALASM TO ABEL™ CONVERTER 6.1

```
TOABEL -linfile -Ooutfile

-linfile      PALASM input file
-Ooutfile     ABEL™ output file
```

TOABEL is a conversion utility that you can use to convert PALASM logic descriptions to ABEL™ logic descriptions. TOABEL creates an ABEL™ source file that can be processed normally with the ABEL™ language processor. This means that the design can be reduced, simulated (if test vectors are supplied), documented and loaded into a logic programmer just as with any other ABEL™ source file.

Note that test vectors for devices with programmable inputs and outputs (i.e., P16L8) may require editing because TOABEL cannot determine whether a pin is an input or an output.

IFLDOC 6.2

ifldoc [-Iin_file] [-Oout_file] [-Ndevice_type]

-Iin_file JEDEC-format input is in *in_file*

-Oout_file output in a Signetics program table format is written to *out_file*

-Ndevice_type*device_type* indicates the type of device described by *in_file*

IFLDOC is a conversion program that converts JEDEC-format files to listing files in a format similar to Signetic Program Logic tables. The listing file created is for documentation purposes only; it will not drive any device programmer.

JEDEC-format input contained in *in_file* is converted to the Signetics-like form and written to *out_file*. The device type described by the JEDEC file must be explicitly specified with the -N parameter. If *in_file* and *out_file* are not specified, input and output are received and sent from the standard input and output devices. A device type must be specified; there is no default value.

For devices with programmable flip-flop types, IFLDOC assumes that the flip-flops are J-K flip-flops unless the Fc line is programmed and the individual FF mode is set to "A". (Refer to IFL logic diagrams for the function and placement of these lines.)

Example:

```
ifldoc -Iifl2.jed -Oifl2.tab -nf82s105
```

This command invokes IFLDOC to convert the JEDEC-format data contained in *ifl2.jed* to a program table format written to *ifl2.tab*. The -N parameter specifies that the device described by *ifl2.jed* is an F82s105. The output file resulting from this command is shown in figure 6-1.

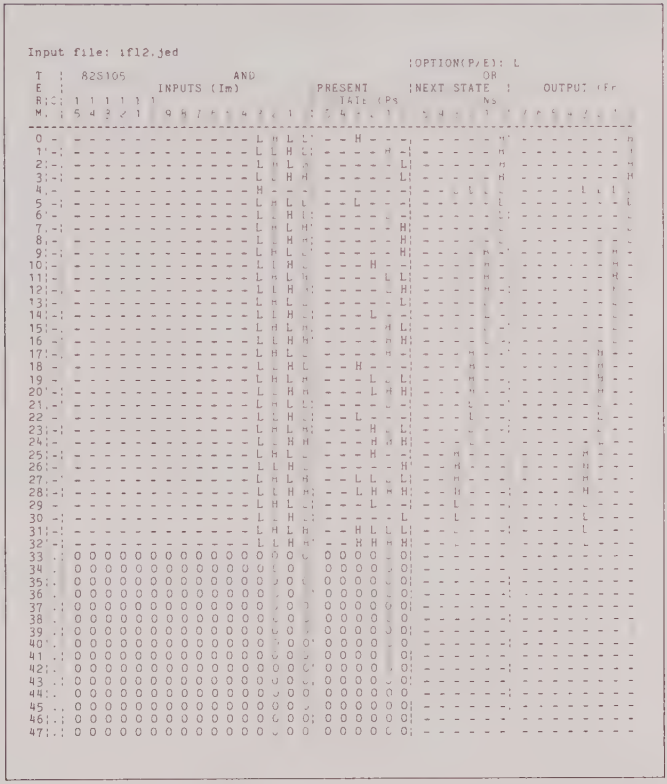


Figure 6-1. Sample Output File From IFLDOC

DEVLIB, DEVICE LIBRARY MANAGER 6.3

devlib [library] [command] { files }

DEVLIB is used to maintain the ABEL™ device library, *abellib.dev*. The device library is a single file that contains all the **.DEV* files (device files) used in earlier versions of ABEL™ and also provided with the current ABEL™ software package. When the ABEL command is invoked, it attempts to find a device file that corresponds to each device referenced in the ABEL™ source file. ABEL™ first looks for the device information in the device library, *abellib.dev*, and if not found looks through the individual **.DEV* files.

The DEVLIB library manager is useful in cases where disk space is at a premium. This program allows you to edit the device library so that unused devices are eliminated from the library.

With DEVLIB, you can add, delete, replace, and extract files from the device library as well as list its contents. The commands available are:

-a add files to the library
-d delete files from the library
-e extract files from the library
-l list the contents of the library

The default library file name is *abellib.dev*. If you do not specify a library name on the command line, this file is referenced by DEVLIB. Note that the *abellib.dev* file will always be used by ABEL™, and there is no provision in ABEL™ for specifying library files with alternate names.

If no commands are specified on the command line, you will be prompted for the function desired.

The suggested procedure to create a library that contains only those devices you will use is:

1. Copy the device files you will require from the ABEL™distribution disk or tape.

2. Rename (or delete from your system) the existing (large) device file library.

3. Use the DEVLIB program to create a new library containing the required files. For example:

   devlib -a file1.dev file2.dev file3.dev ... fileN.dev

4. Install the new library file into the abeldev directory.

ERROR MESSAGES A

This appendix contains listings of the error messages you may encounter while using the ABEL™ language processor or any of the utilities provided with the ABEL™ software package. The error messages are divided into four subsections:

A.1 General Error Messages
A.2 Non-fatal Simulation Error Messages
A.3 TOABEL Error Messages
A.4 IFLDOC Error Messages

Within the subsections, the actual error messages appear in italics. Most of the error messages are listed with a brief description; however those that are self-explanatory are just listed. The symbol 'xxx' in an error message indicates that some actual text or numeric value appears in that location in the message. That text or information may vary depending on the condition causing the error.

Additional information regarding a particular error message can usually be obtained by using the index located at the back of the manual.

GENERAL ERROR MESSAGES A.1

The following error messages may be displayed on the screen during execution of the ABEL™ programs.

Command Line Errors A.1.1

Command line errors indicate that an error was made on the command line when one of the language processor programs was invoked. These errors can occur when a program is invoked directly by you, or from a batch file.

Illegal argument –x
Illegal breakpoint
Illegal option 'xxx', option must start with a '–'
Illegal type of output file 'xxx', valid = 0, 82, 83, 87, or 89
Illegal value for checksum 'xxx', valid = 0, 1 or 2
Illegal value for flag –Xn
Illegal value for unused terms 'xxx', valid = y,n
Input and output file names may not be the same
List and input file names may not be the same
List and output file names may not be the same
Different files must be used for input and output in order to avoid writing over any information.
Unable to open input file 'xxx'
The input file could not be found
Unable to open output file 'xxx'.
The disk is full or contains insufficient space.
'xxx' default X values are 0 or 1.
'xxx' default Z values are 0 or 1.
'xxx' preload options are 0, 1, 2
'xxx' reduction choices are 0, 1, 2, or 3.
'xxx' trace levels are 0, 1, 2, 3, or 4.
'xxx' trace levels for breakpoint are 0, 1, 2, 3, or 4.
'xxx' unrecognized parameter
No device specified
When running the Simulate path, using a JEDEC file as the input, a device type must be specified with the -N option.

Fatal Errors A.1.2

Fatal errors are errors that cause the program that is currently operating to terminate. The error must be corrected before the program will run.

Can't find device file 'xxx'
The specified device file cannot be found in the abellib.dev library file, or in the directory specified with the abeldev variable, or the –Y flag.

Equation for 'xxx' needs reduction
Use the next highest level of reduction to eliminate this error.

Equation for 'xxx' is not sums–of–products
There are two reasons for these errors:
1. The input file to FUSEMAP is not the output file from REDUCE.
2. You used operators other than &, # and ! (and $ for PROMs) and did only a "–R0" reduction in REDUCE. Run REDUCE with a stronger reduction level.

Error writing 'xxxx' file
The output file can no longer be written to. This can arise when a disk is too full, or if the disk data has been damaged.

Expected a '' in JEDEC file*

Fuse out of range in JEDEC file

Illegal fuse state in JEDEC file 'xxx'
The JEDEC file used as input to Simulate contains invalid data or fields.

Illegal operator in equation for 'xxx'
Use the next highest level of reduction to eliminate this error.

Illegal term element at 'xxx'
Use the next highest level of reduction to eliminate this error.

Memory overflow

Memory overflow can occur for any of the following reasons:

1. Too many equations or test vectors.
2. Equations are too complex.
3. Equations have too many $ operators.
4. Insufficient free memory available.
5. Too many symbols.

(Refer to section 3.6 in the Applications Guide for information on reducing the number of product terms.)

Number too large in JEDEC file

This error is generated when reading the JEDEC file with the –N parameter.

PALs and FPLAs can only be output to a JEDEC file

The output file type must be specified with the "–D0" option (JEDEC format) or left unspecified for PALs and FPLAs.

Premature end of source file

PROMs can not be output to a JEDEC file

For PROMs, the output file type must be specified with the "–Dxx" option where xx is not 0.

Too many errors

PARSE terminates execution when many errors are detected in a source file.

Vector in JEDEC file wrong size

Generated when reading JEDEC file using –N flag with wrong device.

Wrong version of intermediate file

Intermediate File Errors A.1.3

Intermediate files are files created by one program in the ABEL™ language processor and used by one or more of the other programs. If there are errors in an intermediate file, it is likely that subsequent programs needing that file will not operate correctly.

File not produced by PARSE
The intermediate file specified as input to TRANSFOR was not produced by PARSE.

File not produced by TRANSFOR
The intermediate file specified as input to REDUCE was not produced by TRANSFOR.

File not produced by FUSEMAP
The intermediate file specified as input to SIMULATE was not produced by FUSEMAP.

File not produced by REDUCE
The intermediate file specified as input to FUSEMAP was not produced by REDUCE

File not produced by ABEL 2.0
The file was not produced by ABEL 2.0 versions of the ABEL programs.

Intermediate file error: TGxxx
An error was detected in the input to TRANSFOR. This may be caused by:
1. An input file generated by PARSE that contained errors.
2. A hardware failure.

Intermediate file error: RGxxx
An error was detected in the input to REDUCE. This may be caused by:
1. An input file generated by PARSE that contained errors.
2. A hardware failure.

Intermediate file error: FGxxx

An error was detected in the input to FUSEMAP. This may be caused by:

1. Using an input file generated by REDUCE that contained errors.
2. A hardware failure.

Intermediate file error: SGxxx

These errors occur in SIMULATE for the following reasons:

1. Someone tampered with the intermediate file.
2. Error in a previous pass.
3. Flag error (i.e. -B1).
4. Hardware failure.

Intermediate file error: DGxxx

These errors in DOCUMENT mean one of the following:

1. The specified input file wasn't created by one of the language processor programs.
2. Someone has tampered with the intermediate file.
3. There was an error in an earlier pass.
4. There is a hardware failure.

Logical Errors A.1.4

Logical errors are device-related errors and point to elements of your design that cannot be implemented on the chosen device, or which are implemented incorrectly.

Logical errors may be created by mistakes in the source file or by errors in your design.

Attribute for 'xxx' is 'latch' but pin is register
This input has the wrong attribute assigned to it. Rewrite your attribute definitions or use a different part.

Attribute for 'xxx' is 'register' but pin is latch
This input has the wrong attribute assigned to it. Rewrite your attribute definitions or use a different part.

Cannot use an XOR for 'xxx'
The XOR operator is not supported for pin 'xxx'. Use a stronger reduction to eliminate the XORs.

Enable for 'xxx' is a pin, not equation
Enable for this pin is not term controllable.

Enable of 'xxx' can only be a constant or a pin
The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Enable of 'xxx' has already been programmed
The enable for this pin is also the enable for other pins and it has already been programmed. Rewrite your equations to eliminate repeat programming of the enable.

Enable of 'xxx' has already been set for this group
The enable for this pin can only be done in groups and the enable for the group that includes this output has already been set. Rewrite your equations so that the enable of this pin is not reprogrammed.

Illegal constant for output 'xxx'
The only allowable constants are 0 and 1: 1 for connecting fuses, and 0 for blowing them.

Illegal pin for enable of 'xxx'
The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Inverted input not allowed from 'xxx'
The pin (or node) 'xxx' does not allow the given type of input. Rewrite your equations to use the other type.

Node number 'xxx' is not defined for this device
You have specified (declared) a pin or node that is not defined for the device. Refer to the logic diagrams supplied with the ABEL™ package to identify device pins and nodes.

Number must be 1 or !0 for enable of 'xxx'
The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Output 'xxx' used more than once
Some pins can be accessed in more than one way. You have equations referencing this pin in more than one way. Rewrite your equations to use only one method.

Pin can only be negative logic for enable of 'xxx'
The enable for this pin can only be programmed in a few ways. Examine the device and change the enable so that it matches the part.

Pin number 'xxx' is not defined for this device
You have specified (declared) a pin or node that is not defined for the device. Refer to the logic diagrams supplied with the ABEL™ package to identify device pins and nodes.

Pin or node 'xxx' is already defined as 'xxx'
You have defined the pin or node with multiple names.

Positive input not allowed from 'xxx'
The input from this pin/node can be only of one type. Rewrite your equations to use the other type.

Signal 'xxx' is not in device 'xxx'
You have specified a pin or node that is not declared in the device.

Signal 'xxx' used or defined as combinational
Signal 'xxx' used or defined as registered
You have specified the wrong equation type ('=' or ':=') for the indicated signal.

Single fuse equation for 'xxx' must be a constant
The only allowable constants are 0 and 1: 1 for connecting fuses, and 0 for blowing them.

Status of 'xxx' register has already been set for this group.
The selection of registered/non-registered pins in this part can only be done in groups, and it has already been done for the group that includes this pin. Rewrite your equations so that all the outputs in this group are of the same type ('=' or ':=').

Too many terms for FPLA array at output 'xxx'
The equation for 'xxx' required too many terms programmed into the FPLA. Use a stronger reduction.

Too many terms for output 'xxx'
There are not enough logic terms in the part to program your equation. Either use a stronger reduction (-r2 for example) or rewrite your equations to use fewer terms.

'xxx' attribute not allowed on this device
You have specified a programmable attribute that is not supported on the device.

'xxx' cannot be enabled
You have written an enable equation for a pin that cannot have an enable equation.

'xxx' cannot be programmed inverted
The equation for 'xxx' has the wrong polarity for this pin and the pin can not be programmed to change polarities. Either rewrite your equations or use a stronger reduction which will automatically convert your equations to the correct polarity.

'xxx' has no enable
'xxx' is pin or node identifier.

'xxx' has no registered feedback
You have used pin 'xxx' as an input or feedback pin but it does not have those features. Rewrite your equations so they do not use this pin as an input.

'xxx' is a registered pin
You have used the wrong type of assignment for this pin. Use ":=" for registered pins and "=" for non-registered pins.

'xxx' is an invalid output
'xxx' is pin or node identifier

'xxx' is not an input pin
Your equation attempts to make use of input or feedback from a pin in the part that is not an input pin and/or has no feedback.

'xxx' is not an input/programmable pin
You have used a pin, that is not an input, as an input in an equation.
'xxx' is not an output pin
You have specified a pin that is not an output pin as an output pin.
'xxx' is not a registered pin
You have used the wrong type of assignment for this pin. Use ":=" for registered pins and "=" for non-registered pins.
'xxx' is not in sums-of-products form
Specify a stronger reduction level and process again.

Preprocessor Errors A.1.5

Preprocessor errors are syntax errors detected by the preprocessor step of PARSE. The preprocessor processes the ABEL™ source file before it is actually parsed; the preprocessor expands macros and acts on directives. To correct a preprocessor error, the source file must be fixed to eliminate the indicated error. The possible error indications are listed below. Refer to the ABEL™ Language Reference Manual for the correct syntax for the BEL™ source files.

'(' expected
';' expected
'=' expected
'xxx' actual arguments expected
Block expected
Constant label expected
Dummy argument expected
Identifier expected
Label expected
Number expected
Number is too large
Radix 'xxx' is not one of 2, 8, 10 or 16
String expected

Syntax Errors A.1.6

Syntax errors are detected by PARSE and indicate missing, incorrect, or incomplete elements and structures in the ABEL™ source file. Correct syntax errors by eliminating the error in the source file. The PARSE listing file may aid you in this. The listing file contains pointers that indicate approximately where in a line the error occurs, followed by the error message describing the error. To correct a syntax error, the source file must be fixed to eliminate the indicated error. The possible error indications are listed below, with a brief explanation where the message may not self-explanatory. Refer to the ABEL™ Language Reference Manual for the correct syntax for the ABEL™ source files.

')' expected
'->' expected
';' expected
':+:' or '::'*
expected
':>' expected
':>' or '->' expected
';' expected
'=' or ':=' expected
']' expected
'THEN' expected
'xxx' actual arguments specified on command line
Number of arguments found after the MODULE keyword is less than that on the command line.
Actual argument length exceeds 'xxx' chars
Bad Element 'xxx' in range
An intermediate identifier name produced as a result of a range is undefined.
Block expected
Can't have letters imbedded in a number
Can't map set onto a different sized set
Can't map set onto a non-set element
Sets cannot be assigned to numbers, signals or special constants; they can only be assigned to other sets.

Cannot operate on signal 'xxx'
Can only operate on constants in this context.
Closing '" of string not found
Closing '}' of block not found
Declaration keyword expected
The valid declaration keywords are:
PIN
NODE
MACRO
DEVICE
ISTYPE
Device_label expected
Digit not in radix 'xxx'
Don't know what device to use
Either an "IN device_id" is necessary or no device was ever specified.
Dummy argument 'xxx' not recognized
The dummy argument found after the "?" was not specified previously.
Element expected
A set, identifier, number, special constant, or parenthesized expression was expected here.
Enables are not registered
'GOTO', 'IF' or 'CASE' expected
Fuse number expected
Identifier expected
Identifier length exceeds 'xxx' chars
Illegal character 'xxx' in source file
'xxx' is the decimal value of the bad character.
Illegal operation on special constant
Illegal set in state diagram header
Check for illegal nested sets in header
Inconsistency in number of parameters
The number of labels on the left side of a declaration keyword did not match the number of declarations on the right side of the declaration keyword.
Industry part number string expected
Invalid flag string 'xxx'
Invalid set range
Keyword 'xxx' is out of context
Label 'xxx' is already defined
Label expected
Max of 'xxx' elements in set was exceeded

Mismatch in number of set elements
Sets with different numbers of elements in them cannot be operated on with a binary operator.

'MODULE' expected

Module label doesn't match 'xxx'
If a label is used after the END statement at the end of a module, that label must match the label used with the MODULE statement that began the module.

Multiple mapping to signal 'xxx'
An attempt was made to assign more than one test condition to a single signal.

Negative declaration not allowed
Negative declarations are only allowed for pins and nodes.

No more than 'xxx' args are declared
More actual arguments are used than there were dummy arguments specified.

Number expected

Numeric overflow
A number which requires more than 32 bits to represent it was specified.

Only ^b, ^o, ^d or ^h radix allowed

Only one label allowed

Operator not allowed

Section keyword or 'END' expected
A section begins with one of the following keywords:

EQUATIONS
TRUTH_TABLE
STATE_DIAGRAM
TEST_VECTORS
FUSES

Signal not allowed

Signal number for 'xxx' is way too large

Signal or .X. expected

Source line length exceeds 'xxx' chars

Special constant must end with a '.'

Special constant not allowed

String expected

String length exceeds 'xxx' chars

Suffix xxx not legal in this context
A shorthand node suffix has been specifed that is not supported for the signal specified.

Undefined compiler directive '@'xxx''
Undefined label 'xxx'
Undefined label 'xxx', maybe 'xxx' was meant
Undefined operation on sets
Undefined operation on signal
Undefined special constant 'xxx'
Undefined token
A character was found outside of a valid context in the source file.
Unrecognized attribute 'xxx'
Unrecognized industry part number 'xxx'
The indicated device is not supported by ABEL™.
Use '#' for OR instead of '|'

Device File/Internal Errors A.1.7.

Device file and internal errors appear as follows:

Device file error: 'xxx'
"xxx" is a number indicating the type of device file error. Any of these errors point to an error in a device specification file, to a hardware error, or possibly to a bug in the program. If a device file error occurs, note the number of the device file error and contact your Data I/O representative.
Internal error: YYxxx
YY is a two character code indicating the program that failed and xxx is the number of the internal error. If a device file error occurs, and all hardware is operating normally, note the number of the device file error and contact your Data I/O representative.

NON–FATAL SIMULATION ERRORS A.2

Non–fatal simulation errors are warnings to you that something is not quite right in your design, but that that error is not severe enough to terminate the simulation. Pay close attention to these errors; they may point to errors in the design that could cause problems in a programmed device.

Device unstable
The device did not stabilize within 20 iterations.

Illegal inputs on RS flip–flop : 'xxx'
See data sheet for correct operation of device. Note the states of nodes controlling flip–flops.

Illegal JEDEC default value : 'xxx'
Errors in reading JEDEC file when using –N flag.

Illegal J–K inputs during load : 'xxx'
See data sheet for correct operation of device. Note the states of nodes controlling flip–flops.

Illegal K input on D flip–flop : 'xxx'
See data sheet for correct operation of device. Note the states of nodes controlling flip–flops.

Illegal P–R inputs during load : 'xxx'
See data sheet for correct operation of device. Note the states of nodes controlling flip–flops.

Output not High–Z during load : 'xxx'
See data sheet for correct operation of device. Note the states of nodes controlling flip–flops.

Unknown JEDEC identifier : 'xxx'
Errors in reading JEDEC file when using –N flag.

TOABEL ERROR MESSAGES A.3

TOABEL errors are shown in numerical rather than alphabetical order.

Fatal error 701 : Invalid command line argument
Fatal error 702 : Duplicate input and output file name
Fatal error 703 : Error opening input file
Fatal error 704 : Error opening output file
Fatal error 705 : Device not supported
Fatal error 706 : Unexpected end of file encountered
Fatal error 707 : Error writing output file

Syntax error 710 : Invalid symbol ':'
Syntax error 711 : Not a valid pin identifier
Syntax error 712 : Duplicate pin identifier
Syntax error 713 : No device found in input file
Syntax error 714 : Missing '('
Syntax error 715 : Too many elements in enable expression
Syntax error 716 : Expected pin identifier
Syntax error 717 : ':' expected
Syntax error 718 : Undefined pin identifier
Syntax error 719 : Invalid symbol in test vector

IFLDOC ERRORS A.4

The following errors may occur during execution of the IFLDOC utility program.

Command Line Errors A.4.1

Too many input options at 'xxx'
This error is caused by specifying more than three input parameters. This also can be caused by having unintentional spaces inside a parameter.

Missing – in 'xxx';
This error occurs if a "–" is not the first character of a parameter entered on the command line.

Unrecognized parameter, 'xxx'
This error occurs if a character other than "i", "I", "o", "O", "n", or "N" is entered as a parameter.

Multiple input files
More than one input file was specified on the command line. A maximum of one is valid.

Multiple output files
More than one output file was specified on the command line. A maximum of one is valid.

Multiple devices
More than one device was specified on the command line. A maximum of one is valid.

No device type specified
A device type must be specified.

Input and output files have the same name
The input and output files must have different names.

Can't open the file named 'xxx'
The input or output file named 'xxx' specified with the –I or –O flags cannot be opened.

Unknown device type, 'xxx'
The device, 'xxx', is not supported, or the device type was entered incorrectly.

Diagnostic Errors A.4.2

Diagnostic errors in IFLDOC indicate that either a hardware failure has occurred or there is a bug in the program. If you can't locate a hardware failure, contact your Data I/O representative. Diagnostic errors are presented in the following form:

Diagnostic Error: 'xxx'

JEDEC Input File Errors A.4.3

Unrecognized field identifier, 'xxx'
This error occurs when an illegal field identifier is encountered in the JEDEC input file.

Missing '' or End of File*
IFLDOC was unable to find a "*" or EOF marker at the end of a JEDEC field.

The JEDEC file contains references to fuses beyond the maximum for this device
This error occurs when the JEDEC file references more fuses than are available in the specified device. For example, if the JEDEC file references fuse number 5000 in a device with only 200 fuses, this error occurs. Check to see that the device type specified on the command line matches the JEDEC file.

Unrecognized character in JEDEC field, 'x'
A character other than "1" or "0" was found in the fuse field.

Default fuse value specified after fuse field reached
To prevent the loss of fuse information, the fuse default value must be specified before any fuse fields ("L" fields) in the JEDEC file. This error occurs when a default value is found after a fuse field.

IFLDOC Warning Messages A.4.4

The following messages are warnings that potential errors may exist in the JEDEC input file, but that those errors are not of a severity that warrants terminating the program.

The number of fuses in this device is not equal to the number given in the JEDEC QF field

You may have the wrong JEDEC file for the device specified on the command line. Check for a match between the JEDEC file and the specified device type.

The JEDEC file may contain less information than is necessary

You may have the wrong JEDEC file for the device specified on the command line. Check for a match between the JEDEC file and the specified device type.

JEDEC FORMAT B

This document defines a format for the transfer of information between a data preparation system and a logic device programmer. This format provides for, but is not limited to, the transfer of fuse, test, identification, and comment information in an ASCII representation. This format defines the "intermediate code" between device programmers and data preparation systems.

Note: This is Data I/O's implementation of the JEDEC (Joint Electron Device Engineering Council) standard No. 3.

BNF RULES AND STANDARD DEFINITIONS B.1

The Backus–Naur Form (BNF) is used in this document to define the syntax of the JEDEC format. BNF is a shorthand notation that follows these rules:

- "::=" denotes "is defined as".
- Characters enclosed by single quotes are literals (required).
- Angle brackets enclose identifiers.
- Square brackets enclose optional items.
- Braces (curly brackets) enclose a repeated item. The item may appear zero or more times.
- Vertical bars indicate a choice between items.
- Repeat counts are given by a *:n* suffix. For example, a six digit number would be defined as "<number> ::= <digit>:6."

For example, in words, the definition of a person's name reads:

> The full name consists of an optional title followed by a first name, a middle name, and a last name. The person may not have a middle name or may have several middle names. The titles consist of: Mr., Mrs., Ms., Miss, and Dr.

The BNF definition is:

```
<full name> ::= [<title>] <f. name>
                {<m. name>} <l. name>
<title> ::= 'Mr.' | 'Mrs.' | 'Ms.' | 'Miss' | 'Dr.'
```

The following standard definitions are used throughout the rest of this document:

```
<digit> ::=  '0'  |  '1'  |  '2'  |  '3'  |  '4'
           |  '5'  |  '6'  |  '7'  |  '8'  |  '9'

<hex-digit> ::= <digit>
              |  'A'  |  'B'  |  'C'  |  'D'  |  'E'  |  'F'

<binary-digit> ::=  '0'  |  '1'

<number> ::= <digit> {<digit>}
```

<del> ::= <space> | <carriage return>

<delimiter> ::= <del> {<del>}

<printable character> ::= <ASCII 20 hex ... 7E hex>

<control character> ::= <ASCII 00 hex ... 1F hex>
| <ASCII 7F hex>

<STX> ::= <ASCII 02 hex>
<ETX> ::= <ASCII 03 hex>
<carriage return> ::= <ASCII 0D hex>
<line feed> ::= <ASCII 0A hex>
<space> ::= <ASCII 20 hex> | ' '

<valid character> ::= <printable character>
| <carriage return>
| <line feed>

<field character> ::= <ASCII 20 hex ... 2A hex>
| <ASCII 2C hex ... 7E hex>
| <carriage return>
| <line feed>

Syntax of the Transmission Protocol

<format> ::= <STX> <design spec> {<field>}
<ETX> <xmit check-sum>

The transmission consists of a start-of-text (STX) character, various fields, an end-of-text (ETX) character, and a transmission check-sum. The character set consists of the printable ASCII characters and four control characters (STX, ETX, CR, LF). Other control characters should not be used because they can produce undesirable side-effects in the receiving equipment.

Figure B-1 shows a complete JEDEC format file for transmission. The individual components of the format are discussed in the following sections.

```
<STX>
Acme Logic Design  Joan Engineer   Feb. 29 1983
Widget Decode  756-AB-3456 Rev C  Device Mullard 12AX7*
QP20* QF384* G1*
L0000 1111111011 1111111111  1111000000 0000000000
      0000000000 0000000000  0000000000 0000000000
      0000000000 0000000101  1111111111 1111111111
      0000000000 0000000000  0000111101 1111111111
      1111111111 1111110111  1111111111 1111111111*

L0200 1110101111 1111110000  0000000000 0000000000
      1111111111 1111011011  1111111111 1111111110
      0111111111 1111111111  1111111110 1111111111
      1111111111 1111101111  1111111111 1111101111
      0000000000 0000000000  0000*       C1BB9*

V0001 XXXXXXXXXNXXXXXXXHON*    V0002 XXXXXXXXXNXXXXXXL1N*
V0003 00XXXXXXXNXXXXXXLXXN*    V0004 01XXXXXXXNXXXXXXLXXN*
V0005 10XXXXXXXNXXXXXXLXXN*    V0006 11XXXXXXXNXXXXXXHXXN*
V0007 XX00XXXXXNXXXXXXLXXXN*   V0008 XX01XXXXXNXXXXXHXXXN*
V0009 XX10XXXXXNXXXXXXHXXXN*   V0010 XX11XXXXXNXXXXXHXXXN*

T02* S10101010101010101010*  R1ACB678F*
<ETX>32EB
```

Figure B-1. Example of a JEDEC Format Transmission File

DESIGN SPECIFICATION B.3

Syntax of the Design Specification

<design spec> ::= {<field character>}'*'

The design specification is the first field in the format, must be included, and does not have an identifier signalling its start. An asterisk terminates the field. The design specification should consist of:

1. User's name and company
2. Date, part number, and revision
3. Manufacturer's device number
4. Other information

TRANSMISSION CHECKSUM B.4

Syntax of the Transmission Checksum

<xmit check-sum> ::= <hex-digit>:4

The transmission check-sum is the 16-bit sum (i.e., modulo 65,535) of all ASCII characters transmitted between and including the STX and ETX. The parity bit is excluded in the calculation.

FIELDS B.5

Syntax of Fields

```
<field> ::= [<delimiter>] <field identifier>
            {<field character>} '*'

<field identifier> ::=  'C' | 'D' | 'F' | 'G'
                      | 'L' | 'M' | 'N' | 'P'
                      | 'Q' | 'R' | 'S' | 'T'
                      | 'V'

<reserved identifier> ::=  'A' | 'B' | 'E' | 'H'
                         | 'I' | 'J' | 'K' | 'O'
                         | 'U' | 'W' | 'X' | 'Y'
                         | 'Z'
```

Each field begins with a single character identifier that identifies the field type. Multiple character identifiers can be used to create sub-fields (i.e., "A1", "A$", or "AB3"). The field is terminated with an asterisk. Therefore, asterisks cannot be imbedded within the field. While not required, carriage returns and line feeds should be used to improve the readability of the format. Reserved identifiers currently have no function and are reserved for future use. The meanings of the field identifiers is given in table B-1.

Table B-1. Field Identifiers

(* indicates reserved for future use)	
A – *	N – Note
B – *	O – *
C – Check sum	P – Pin sequence
D – Device type	Q – Value
E – *	R – Resulting vector
F – Default fuse state	S – Starting vector
G – Security fuse	T – Test cycles
H – *	U – *
I – *	V – Test vector
J – *	W – *
K – *	X – *
L – Fuse list	Y – *
M – Option	Z – *

Device Field (D) B.5.1

Syntax of the device field:

<device> :: = 'D' <field characters> '*'

The device field defines the programmable logic device to be programmed.

Example:

D9501*

Fuse Information Fields (F, L, C) B.5.2

Syntax of the fuse information fields:

```
<fuse information> ::= [<default state>] <fuse list>
                       {<fuse list>}
                       [<fuse checksum>]

<fuse list> ::= 'L' <number> <delimiter>
                {<binary-digit> [<delimiter>]} '*'

<default state> ::= 'F' <binary-digit> '*'

<fuse checksum> ::= 'C' <hex-digit>:4 '*'
```

Each fuse of a device is assigned a decimal number and has two possible states: a zero, specifying a low resistance link, or a one, specifying a high resistance link. Fuse information describing the state of each fuse in the device is given by three fields: the fuse list (L field), the default state (F field), and the fuse checksum (C field).

Fuse states are explicitly defined by the L field. The character *L* begins the L field and is followed by the number of the first fuse for which this field defines a state. The first fuse number is followed by a list of binary values (0 or 1) that indicate the fuse states. When more than one binary value is specified, the additional values are assigned to fuses numbered consecutively from the first fuse number. The L field can be any length desired, and any number of L fields can be specified. If the state for a fuse is specified more than once, the last state replaces all previous ones specified for that fuse.

The F field defines the states of fuses that are not explicitly defined in the L fields. If no F field is specified, all fuse states must be defined by L fields.

The fuse information checksum field is used to detect transmitting and receiving errors. The field contains a number computed by adding 8-bit words containing the fuse states for a device. The 8-bit words are formed as shown in figure B-2. Unused bits in the final 8-bit word are set to zero before the checksum is calculated.

word 00	|msb|							lsb
Fuse No.	7	6	5	4	3	2	1	0
word 01	|msb|							lsb
Fuse No.	15	14	13	12	11	10	9	8
word 62	|msb|							lsb
Fuse No.	–	–	–	–	499	498	497	496

Figure B-2. 8-Bit Words Formed From Fuse States for Checksum

Following is an example of full specification of the L, C, and F fields:

```
F0*L0 01010101* L0008 01010111*
L1000 0101*CF3BA*
```

Another example, where F and C are not specified:

```
L0200  01101010101010101011
0101110101101000010010010*
```

Structured Functional Test Information (V, P fields) B.5.3

Syntax of Functional Test Information:

```
<function test> ::= [<pin list>] <test vector>
                    {<test vector> }

<pin list> ::= 'P' <pin number>:N '*'

<pin number> ::= <delimiter> <number>

N ::= number of pins on device

<test vector> ::= 'V' <number> <delimiter>
                  <test condition>:N '*'

<test condition> ::= <digit> 'C' | 'F' | 'H'
                   | 'K' | 'L' | 'N' | 'P'
                   | 'X' | 'Z'

<reserved condition> ::= 'A' | 'B' | 'D' | 'E'
                       | 'G' | 'I' | 'J' | 'M'
                       | 'O' | 'Q' | 'R' | 'S'
                       | 'T' | 'U' | 'V' | 'W'
                       | 'Y' | 'Z'
```

Functional test information is specified by test vectors containing test conditions for each device pin. Each test vector contains *n* test conditions where *n* is the number of pins on the device. Table B–2 lists the conditions that can be specified for device pins.

The C and K driving signals are presented after the other inputs are stable. The L, H, and Z tests are performed after all inputs have stabilized, including C and K.

Test vectors are numbered by following the *V* character with a number. The vectors are applied in numerical order to the device being tested. If the same numbered vector is specified more than one time, the data in the last vector replaces any data contained in previous vectors with that number.

Table B–2. Test Conditions

0	–	Drive input low
1	–	Drive input high
2–9	–	Drive input to supervoltage # 2–9
C	–	Drive input low, high, low
F	–	Float input or output
H	–	Test output high
K	–	Drive input high, low, high
L	–	Test output low
N	–	Power pins and outputs not tested
P	–	Preload registers
X	–	Output not tested, input undefined
Z	–	Test input or output for high impedance

The conditions contained in test vectors are applied to the device pins in numerical order from left to right unless specified otherwise with the P field. (The leftmost condition is applied to pin 1, and the rightmost condition is applied to pin 20 of a 20 pin device, for example.) The P field indicates an alternate correspondence between the test conditions and the pin numbers.

The following example uses both the V and P fields to specify functional tests information for a device:

```
P 11 12 13 14 15 16 17 18 19 20
  10  9  8  7  6  5  4  3  2  1*

V0001 C01010101NHLLLHHLHLN*
V0002 C01011111NHLLHLLLHLN*
V0003 C10010111NZZZZZZZZZN*

V4 C01010100NFLHHLFFLLN*
```

Optional Information Fields (G, S, R, M, N, Q, T) B.5.4

Syntax of Optional Information Fields:

```
<option field> ::= <option ident.>
                   {<field character>} '*'

<option ident.> ::= 'G' | 'S' | 'R' | 'M'
                  | 'N' | 'Q' | 'T'
```

Optional information can be defined using the G, S, R, M, N, Q, and T fields. Each field must begin with the appropriate character and end with an asterisk; no other restrictions exist. Three examples of optional information fields are given here:

```
Q*
MFG Acme Semiconductor*
M:1234*
```

Data I/O uses six optional fields: three for the Logic Fingerprint™ test, a security fuse field, a value field, and a note field.

LOGIC FINGERPRINT™ (S, R, T)

Syntax for S, R, T

```
<starting vector> ::= 'S' <test condition>:N '*'
<resulting vector> ::= 'R' <hex-digit>:8 '*'
<test cycles> ::= 'T' <number> '*'

N ::= number of pins on device
```

Logic Fingerprint™ tests are specified by the S, R, and T fields. The S field defines the starting vector for the Logic Fingerprint™ test. The possible states are 0 (TTL low) and 1 (TTL high). The R field contains the resulting vector or test-sum. The T field denotes the number of test cycles to be run.

Example:

```
S01000100001110001111O110*
R5BCD34A7*
T01*
```

SECURITY FUSE (G)

Syntax for the Security Fuse Field

<security fuse> ::= 'G' <binary-digit> '*'

The security fuses of certain logic devices may be enabled for programming by sending a 1 in the G field.

Example:

G1*

VALUES (QF, QP)

Syntax for QF, QP:

<number of pins> ::= 'QP' <number> '*'
<fuse limit> ::= 'QF' <number> '*'

The Q field expresses values or limits required by the receiving device. Two subfields are defined: the P subfield for number of pins on the device, and the F subfield for the number of fuses. These two values enable the receiving device and process the other fields without knowing the device manufacturer or family/pinout code.

Example:

QP24* QF1024 *

NOTE FIELD (N)

Syntax of the Note Field:

<note> ::= 'N' <field characters> '*'

The note field is used to place notes and comments in the transfer file.

Example:

N Test Preload*

OTHER RULES B.6

This section discusses other rules and restrictions concerning the JEDEC standard. Variations on the standard are also discussed.

Transportability B.6.1

All receiving machines should have a "kernel" mode to ignore all optional fields so that the actual programming data will be transportable. For example, optional fields can be sent to specify additional checksums. A receiving machine in the "kernel" mode could ignore this information, yet still receive the link information required to program the device.

If the optional F field is used to avoid transmitting fuse state data, transportability could be lost. Therefore, whenever practical, data should be transmitted for all links of the device (by using the L field).

Some computer operating systems add control characters after each line making it very difficult to compute the transmission checksum. To disable the transmission checksum, receiving equipment should accept "0000" as a valid checksum.

Syntax for Minimum Transmission to Enhance Portability:

```
<kernel> ::= <STX> <design spec>
             <min. fuse information> <ETX>
             <xmit check-sum>

<design spec> ::= {<field character>}'*'
<min. fuse information> ::= <fuse list> {<fuse list>}
```

An example of a "kernel" JEDEC transmission is shown in figure B-3.

Variation From Present JEDEC Standard by Data I/O B.6.2

1. The delimiter after the link number or the vector number may be any combination of carriage returns, line feeds, and/or spaces.

2. The checksum and sumcheck are renamed fuse checksum and xmit checksum respectively.

Variations for 303A LogicPak™ V01 B.6.3

1. The link number (L field) and the vector number (V field) must both be 4-digit numbers.

2. The delimiter after the link number and the vector number must include a space.

3. Spaces are not allowed between the terminating asterisk of one field and the identifier of the next field.

4. The security fuse ("G" field) requires a space after the binary digit.

5. The kernel mode is not implemented.

6. The D field must have valid family and pinout codes.

   <device> ::= 'D' <hex-digit>:4 '*'

```
<STX>
Acme Logic Design  Jane Engineer   Feb. 29 1983
Widget Decode  756-AB-3456 Rev C  Device Mullard 12AX7*

L0000 1111111011 1111111111  1111000000 0000000000
      0000000000 0000000000  0000000000 0000000000
      0000000000 0000000101  1111111111 1111111111
      0000000000 0000000000  0000111101 1111111111
      1111111111 1111110111  1111111111 1111111111*

L0200 1110101111 1111110000  0000000000 0000000000
      1111111111 1111011011  1111111111 1111111110
      0111111111 1111111111  1111111110 1111111111
      1111111111 1111101111  1111111111 1111101111
      0000000000 0000000000  0000*

<EXT>0000
```

Figure B-3. "Kernel" Transmission File

Table C-1 lists the nodes used by ABEL™ along with the names given to those nodes by the manufacturer. Refer also to the manufacturer's logic diagrams or those supplied in Data I/O's Programmable Logic Device Schematics publication. Device names are given in boldface followed by the number of pins on the device; node numbers follow. The use of nodes is discussed in the ABEL™ Language Reference and Applications Guide.

Table C-1. Nodes Used by ABEL™

Node No.	Shorthand Notation	Name	Function
F82S105 (28 Pins)			
29	.R	RF0	R f/f input for F0 (pin 18)
30	.R	RF1	R f/f input for F1 (pin 17)
31	.R	RF2	R f/f control for F2 (pin 16)
32	.R	RF3	f/f input for F3 (pin 15)
33	.R	RF4	R f/f input for F4 (pin 13)
34	.R	RF5	R f/ input for F5 (pin 12)
35	.R	RF6	R f/f input for F6 (pin 11)
36	.R	RF7	R f/f input for F7 (pin 10)
37	—	P0	S f/f input (and output) for P0
38	—	P1	S f/f input (and output) for P1
39	—	P2	S f/f input (and output) for P2
40	—	P3	S f/f input (and output) for P3
41	—	P4	S f/f input (and output) for P4
42	—	P5	S f/f input (and output) for P5
43	.R	RP0	R f/f input for P0
44	.R	RP1	R f/f input for P1
45	.R	RP2	R f/f input for P2
46	.R	RP3	R f/f input for P3
47	.R	RP4	R f/f input for P4
48	.R	RP5	R f/f input for P5
49	—	COMP	Complement term at offset 44

Table C-1. Nodes Used by ABEL™ (Continued)

Node No.	Shorthand Notation	Name	Function
F82S155 (20 Pins)			
21	—	FC	f/f mode control
22	.PR	PB	B group f/f preset
23	.RE	RB	B group f/f reset
24	.PR	PA	A group f/f preset
25	.RE	RA	A group f/f reset
26	.L	LB	B group f/f load
27	.L	LA	A group f/f load
28	—	COMP	complementary node at offset 32
29	.M	M0	f/f mode control for F0 (pin 14)
30	.M	M1	f/f mode control for F1 (pin 15)
31	.M	M2	f/f mode control for F2 (pin 16)
32	.M	M3	f/f mode control for F3 (pin 17)
33	.K	K0	K f/f input for F0 (pin 14)
34	.K	K1	K f/f input for F1 (pin 15)
35	.K	K2	K f/f input for F2 (pin 16)
36	.K	K3	K f/f input for F3 (pin 17)
37	.J	K4	J0 f/f alt. input for F0 (pin 14)
38	.J	J1	J f/f alt. input for F1 (pin 15)
39	.J	J2	J f/f alt. input for F2 (pin 16)
40	.J	J3	J f/f alt. input for F3 (pin 17)

Table C–1. Nodes Used by ABEL™ (Continued)

Node No.	Shorthand Notation	Name	Function
F82S157 (20 Pins)			
21	—	FC	f/f mode control
22	.PR	PB	B group f/f preset
23	RE	RB	B group f/f reset
24	.PR	PA	A group f/f preset
25	.RE	RA	A group f/f reset
26	.L	LB	B group f/f load
27	.L	LA	A group f/f load
28	—	COMP	complement node at offset 32
29	.M	M0	f/f mode control for F0 (pin 13)
30	.M	M1	f/f mode control for F1 (pin 14)
31	.M	M2	f/f mode control for F2 (pin 15)
32	.M	M3	f/f mode control for F3 (pin 16)
33	.M	M4	f/f mode control for F4 (pin 17)
34	.M	M5	f/f mode control for F5 (pin 18)
35	.K	K0	K f/f input for F0 (pin 13)
36	.K	K1	K f/f input for F1 (pin 14)
37	.K	K2	K f/f input for F2 (pin 15)
38	.K	K3	K f/f input for F3 (pin 16)
39	.K	K4	K f/f input for F4 (pin 17)
40	.K	K5	K f/f input for F5 (pin 18)
41	.J	J0	J f/f alt. input for F0 (pin 13)
42	.J	J1	J f/f alt. input for F1 (pin 14)
43	.J	J2	J f/f alt. input for F2 (pin 15)
44	.J	J3	J f/f alt. input for F3 (pin 16)
45	.J	J4	J f/f alt. input for F4 (pin 17)
46	.J	J5	J f/f alt. input for F5 (pin 18)

Table C-1. Nodes Used by ABEL™ (Continued)

Node No.	Shorthand Notation	Name	Function
F82S159 (20 Pins)			
21	—	FC	f/f mode control
22	.PR	PB	B group f/f preset
23	.RE	RB	B group f/f reset
24	.PR	PA	A group f/f preset
25	.RE	RA	A group f/f reset
26	.L	LB	B group f/f load
27	.L	LA	A group f/f load
28	—	COMP	complement node at offset 32
29	.M	M0	f/f mode control for F0 (pin 12)
30	.M	M1	f/f mode control for F1 (pin 13)
31	.M	M2	f/f mode control for F2 (pin 14)
32	.M	M3	f/f mode control for F3 (pin 15)
33	.M	M4	f/f mode control for F4 (pin 16)
34	.M	M5	f/f mode control for F5 (pin 17)
35	.M	M6	f/f mode control for F6 (pin 18)
36	.M	M7	f/f mode control for F7 (pin 19)
37	.K	K0	K f/f input for F0 (pin 12)
38	.K	K1	K f/f input for F1 (pin 13)
39	.K	K2	K f/f input for F2 (pin 14)
40	.K	K3	K f/f input for F3 (pin 15)
41	.K	K4	K f/f input for F4 (pin 16)
42	.K	K5	K f/f input for F5 (pin 17)
43	.K	K6	K f/f input for F6 (pin 18)
44	.K	K7	K f/f input for F7 (pin 19)
45	.J	J0	J f/f alt. input for F0 (pin 12)
46	.J	J1	J f/f alt. input for F1 (pin 13)
47	.J	J2	J f/f alt. input for F2 (pin 14)
48	.J	J3	J f/f alt. input for F3 (pin 15)
49	.J	J4	J f/f alt. input for F4 (pin 16)

Table C-1. Nodes Used by ABEL™ (Continued)

Node No.	Shorthand Notation	Name	Function
50	.J	J5	J f/f alt. input for F5 (pin 17)
51	.J	J6	J f/f alt. input for F6 (pin 18)
52	.J	J7	J f/f alt. input for F7 (pin 19)
F82S167 (24 Pins)			
25	.R	RF0	R f/f input for F0 (pin 9)
26	.R	RF1	R f/f input for F1 (pin 10)
27	.R	RF2	R f/f input for F2 (pin 11)
28	.R	RF3	R f/f input for F3 (pin 13)
29	.R	RP0	R f/f input for P0 (pin 14)
30	.R	RP1	R f/f input for P1 (pin 15)
31	—	P7	S f/f input (and output) for P7
32	—	P6	S f/f input (and output) for P6
33	—	P5	S f/f input (and output) for P5
34	—	P4	S f/f input (and output) for P4
35	—	P3	S f/f input (and output) for P3
36	—	P2	S f/f input (and output) for P2
37	.R	RP7	R f/f input for P7
38	.R	RP6	R f/f input for P6
39	.R	RP5	R f/f input for P5
40	.R	RP4	R f/f input for P4
41	.R	RP3	R f/f input for P3
42	.R	RP2	R f/f input for P2
43	—	COMP	Complement term at offset 45
EP300			
21	—	R	Reset term at fuse 2628
22	—	P	Preset term at fuse 2592
P22V10 (24 Pins)			
25	—	R	Reset term at fuse 0000
26	—	P	Preset term at fuse 5764

LANGUAGE REFERENCE

DATA I/O

TABLE OF CONTENTS

SECTION 1. INTRODUCTION

1.1 OTHER ABEL™ MANUALS 1-1
1.2 NOTATION CONVENTIONS 1-2

SECTION 2. LANGUAGE ELEMENTS

2.1 BASIC SYNTAX 2-1
2.2 VALID ASCII CHARACTERS 2-2
2.3 IDENTIFIERS 2-3
2.3.1 Reserved Identifiers 2-4
2.3.2 Choosing Identifiers 2-5
2.4 STRINGS 2-6
2.5 COMMENTS 2-7
2.6 NUMBERS 2-8
2.7 SPECIAL CONSTANT VALUES 2-10
2.8 OPERATORS, EXPRESSIONS, AND EQUATIONS 2-11
2.8.1 Logical Operators 2-11
2.8.2 Arithmetic Operators 2-12
2.8.3 Relational Operators 2-13
2.8.4 Assignment Operators 2-15
2.8.5 Expressions 2-16
2.8.6 Equations 2-18
2.9 SETS 2-20
2.9.1 Set Operations 2-21
2.9.2 Set Assignment and Comparison 2-24
2.9.3 Restrictions on Sets 2-25
2.10 BLOCKS 2-26
2.11 ARGUMENTS AND ARGUMENT SUBSTITUTION 2-27

SECTION 3. STRUCTURE

3.1 BASIC STRUCTURE 3-1
3.2 MODULE STATEMENT AND STRUCTURE 3-3
3.3 FLAG STATEMENT 3-5
3.4 TITLE STATEMENT 3-6

3.5 DECLARATIONS 3-7
3.5.1 Device Declaration Statement 3-7
3.5.2 Pin Declaration Statement 3-8
3.5.3 Node Declaration Statement 3-10
3.5.4 Constant Declaration Statement 3-12
3.5.5 Macro Declaration Statement and Macro Expansion 3-14
3.5.6 ISTYPE Statement 3-17
3.6 EQUATIONS STATEMENT 3-21
3.7 TRUTH TABLES 3-22
3.7.1 Truth Table Header Syntax 3-22
3.7.2 Truth Table Format 3-24
3.8 STATE DIAGRAMS 3-26
3.8.1 STATE_DIAGRAM Statement 3-26
3.8.2 IF-THEN-ELSE Statement 3-29
3.8.3 CASE Statement 3-31
3.8.4 GOTO Statement 3-32
3.9 FUSES SECTION 3-35
3.10 TEST VECTORS 3-36
3.10.1 Test Vector Table Format 3-37

SECTION 4. DIRECTIVES

4.1 @ALTERNATE Directive 4-2
4.2 @CONST (Constant) Directive 4-3
4.3 @EXIT Directive 4-3
4.4 @EXPR (Expression) Directive 4-4
4.5 @IF Directive 4-4
4.6 @IFB (If Blank) Directive 4-5
4.7 @IFDEF (If Defined) Directive 4-5
4.8 @IFIDEN (If Identical) Directive 4-6
4.9 @IFNB (If Not Blank) Directive 4-6
4.10 @IFNDEF (If Not Defined) Directive 4-7
4.11 @IFNIDEN (If Not Identical) Directive 4-7
4.12 @INCLUDE Directive 4-8
4.13 @IRP (Indefinite Repeat) Directive 4-9
4.14 @IRPC (Indefinite Repeat, Character) Directive . 4-10
4.15 @MESSAGE Directive 4-11
4.16 @PAGE Directive 4-11
4.17 @RADIX Directive 4-12
4.18 @REPEAT Directive 4-13
4.19 @STANDARD Directive 4-13

APPENDIX A. SYNTAX DIAGRAMS

A.1 How to Read Syntax Diagrams A-1
A.2 ABEL™ Syntax Diagrams A-3

APPENDIX B. ASCII TABLES

APPENDIX C. SET OPERATIONS

FIGURES

1-1 Sample Syntax Description 1-5
3-1 Structure of an ABEL™ Source File 3-2
3-2 Block Diagram and Timing Diagram for a Latch 3-19
3-3 Feedback Paths for an EP300 3-20
3-4 Pictorial State Diagram 3-24

TABLES

1-1 Sections and Contents of the Language Reference 1-2
1-2 Notation Conventions 1-4
2-1 Valid ASCII Characters 2-2
2-2 Reserved Identifiers 2-4
2-3 Number Representation in Different Bases 2-8
2-4 Special Constant Values 2-10
2-5 Logical Operators 2-11
2-6 Arithmetic Operators 2-12
2-7 Relational Operators 2-13
2-8 Assignment Operators 2-15
2-9 Summary of Operators and Priorities 2-17
2-10 Valid Set Operations 2-21
4-1 Directives 4-1
4-2 Alternate Operator Set 4-2

1 INTRODUCTION

The Language Reference manual describes in detail the elements and structure of ABEL™. It is intended both as a complete specification of the language to be used by the new ABEL™ designer and as a quick reference for the more experienced designer.

The manual is divided into four major sections as shown in table 1–1.

Table 1–1. Sections and Contents of the Language Reference

Section	Title	Description
1	Introduction	Introduction to the manual
2	Elements	Describes the basic elements of ABEL™, such as keywords, constants, expressions, equations and sets.
3	Structure	Explains how the various elements are incorporated into different structures to form an ABEL™ logic description.
4	Directives	Discusses directives that influence processing of the source file.

OTHER ABEL™ MANUALS 1.1

The Language Reference is one of three documents describing ABEL™. The other two manuals are:

ABEL™ User's Guide

This manual describes how to use the ABEL™ language processor to convert logic descriptions to programmer load files. Setup procedures are described and a sample session with the language processor is given. All processor options are described in detail and the output files are explained. The procedure for downloading programmer load files is presented.

ABEL™ Applications Guide

This manual presents information and examples to help the designer use ABEL™. Specifications, design methods, and complete source files are given for typical programmable logic designs. Advanced designs are presented. Suggestions and tips concerning the use of ABEL™ are given.

NOTATION CONVENTIONS 1.2

Table 1-2 shows notation conventions used in the definitions and syntax descriptions in this manual.

Table 1-2. Notation Conventions

Notation	Usage
italics	Indicates references by name to items contained in examples, figures, listings and tables.
quotation marks (")	Surround italicized items when the italicized reference contains spaces.
UPPER-CASE	In syntax descriptions and diagrams, indicates that the word or letters in upper case must be entered in full. The item can be entered in either upper-case or lower-case. Indicates a keyword.
lower-case	In syntax descriptions and diagrams, indicates that a value or name is supplied by the user in place of the word or letters in lower-case. The user-supplied value can be entered in either upper-case or lower-case.
square brackets [ ]	Surround optional entries that can be supplied or omitted as necessary.
ellipsis ...	Indicates that the preceding item can be repeated as necessary.
all other punctuation	Apostrophes, exclamation points, parentheses, quotes and commas must be entered exactly as shown.

An example of italics used in text:

> In table 1-2, the word *ellipsis* refers to three periods in a row.

An example of quotation marks used in text:

> If the keyword had been entered as *"Mod ule"*, it would have been interpreted as two identifiers, *Mod* and *ule*.

An example of upper-case, lower-case, square brackets, and ellipses as used in a syntax description:

> MODULE modname [(arg [,arg]...)]

This indicates that the keyword, *MODULE*, must be entered, but it can be entered in either upper-case or lower-case. A name for the module must be entered in place of *modname*. None or some arguments can be supplied in place of *arg*. If any arguments are supplied, they must be surrounded by parentheses. The following would be a valid user entry:

> module ADDER (CarryIn,add1,ADD2)

The definition of ABEL™ syntax is given with the help of syntax descriptions. A syntax description contains a complete and formal definition of the syntax of a language construct and definitions of the individual pieces that make up the construct. The first line of a syntax description (shown in boldface) defines the syntax according to the notation conventions listed above. The lines following the first define the elements of that syntax. Syntax descriptions are contained within shaded boxes. Figure 1-1 shows a sample syntax description.

Figure 1-1. Sample Syntax Description

device_id [,device_id]... DEVICE real_device ;

device_id	an identifier used in a module for references to a device.
real_device	a string describing the industry part number of the real device represented by *device_id*.

LANGUAGE ELEMENTS 2

This section describes the various elements of the ABEL™ design language. These language elements are combined according to the structure described in section 3 to create ABEL™ logic descriptions. Each element is presented on the following pages. The organization is from the most basic to the most complex topics.

BASIC SYNTAX 2.1

Each line in an ABEL™ source file must conform with the following syntax rules and restrictions:

1. A line may be up to 131 characters long.

2. Lines are ended by a line feed character (hex 0A), by a vertical tab (hex 0B), or by a form feed (hex 0C). Carriage returns in a line will be ignored, thus accommodating common end-of-line sequences, such as carriage return/line feed. On most computers, an input line is ended simply by pressing a RETURN or ENTER key.

3. Keywords, identifiers, and numbers must be separated from each other by at least one space. Exceptions to this rule are in lists of identifiers separated by commas, in expressions where identifiers or numbers are separated by operators, or in places where parentheses provide the separation.

4. Spaces cannot be imbedded in the middle of keywords, numbers, operators or identifiers. Spaces can appear in strings, comments, blocks and actual arguments. For example, if the keyword MODULE is entered as "*MOD ULE*", it will be interpreted as two identifiers, *MOD* and *ULE*. Similarly, if you enter "102 05", meaning the number 10205, it will be interpreted as two numbers, 102 and 5.

5. Keywords (words defined as part of the language and that have specific uses) can be typed in either upper-case or lower-case with no difference in effect.

6. Identifiers (user-supplied names and labels) can be typed in either upper-case, lower-case, or mixed-case, but are case-sensitive: the identifier *output*, typed in all lower-case letters, is not the same as the identifier *Output* with an upper-case "O".

VALID ASCII CHARACTERS 2.2

All upper-case and lower-case alphabetic characters and most other characters on common keyboards are valid. Table 2-1 shows all the valid characters.

Table 2-1. Valid ASCII Characters

a - z	lower-case alphabet
A - Z	upper-case alphabet
0 - 9	digits
space	
tab	

!	@	#	$	%	^	&	*	(	)	-	
_	=	+	[	{	]	}	;	:	'	"	
`	~	\	|	,	<	>	.	/	?		

IDENTIFIERS 2.3

Identifiers are names that identify devices, device pins or nodes, sets, input or output signals, constants, macros, and dummy arguments. All of these items are defined later in section 2. The rules and restrictions for identifiers are the same regardless of what the identifier describes.

The rules governing identifiers are:

1. Identifiers must begin with an alphabetic character or with an underscore.

2. Identifiers may be up to 31 characters long. Anything longer than 31 characters is considered an error and is flagged by the language processor.

3. Other than the first character (see rule 1), identifiers may contain upper-case and lower-case alphabetic characters, digits and underscores.

4. Spaces cannot be used in an identifier. Use underscores to provide separation between words.

5. Identifiers are case-sensitive: upper-case letters and lower-case letters are not the same.

Some valid identifiers are:

```
HELLO
hello
__K5input
P__h
This__is__a__very__long__identifier
```

Note the use of underscores to separate words. Note also that lower-case letters are not the same as upper-case letters. Thus *hello* is a different identifier than *HELLO*. Use different cases to make your source file easy to read.

Some invalid identifiers are:

7_	Does not begin with a letter or underscore
$4	Does not begin with a letter or underscore
HEL.LO	Contains a period
b6 kj	Contains a space

Note that the last of these invalid identifiers will be seen by the language processor as two identifiers, *b6* and *kj*.

Reserved Identifiers 2.3.1

Keywords are reserved identifiers that are parts of the ABEL™ Design Language. This means that they cannot be used to name devices, pins, nodes, constants, sets, macros or signals. When a keyword is used, it refers only to the function of that keyword. If a keyword is used in the wrong context, an error is flagged by the language processor. Table 2–2 shows the reserved identifiers.

Table 2–2. Reserved Identifiers

CASE	FUSES	PIN
DEVICE	GOTO	STATE
ELSE	IF	STATE_DIAGRAM
ENABLE	IN	TEST_VECTORS
END	ISTYPE	THEN
ENDCASE	LIBRARY	TITLE
ENDWITH	MACRO	TRUTH_TABLE
EQUATIONS	MODULE	WITH
FLAG	NODE	

Choosing Identifiers 2.3.2

The right choice in identifiers can make a source file easy to read and understand. This is especially important in an environment where more than one person may work on the same project or where one designer will take up another's earlier work.

The following suggestions are given to help make logic descriptions self-explanatory, thus eliminating the need for extensive documentation.

- Choose identifiers that match the function of that which they describe. For example, the pin to be used as the carry-in on an adder could be named *Carry_In*. For a simple OR gate, the two input pins might be given the identifiers *IN1* and *IN2*, and the output might be named *OR*.
- Avoid large numbers of similar identifiers. For example, do not name the outputs of a 16 bit adder *ADDER_OUTPUT_BIT_1*, *ADDER_OUTPUT_BIT_2*, and so on. Such grouping of names makes the source file difficult to read.
- Use underscores to separate words in your identifier. *THIS_IS_AN_IDENTIFIER* is much easier to read than *THISISANIDENTIFIER*.
- Mixed-case identifiers can help make your source file readable: *CarryIn*

STRINGS 2.4

Strings are series of ASCII characters enclosed by single quotes (apostrophes). Strings are used in the TITLE, MODULE and FLAG statements, and in pin, node, and attribute declarations. Spaces are allowed in strings.

Valid strings:

```
'Hello'
'  Text with a space in front'
''
'The preceding line is an empty string'
'Punctuation? is even allowed!!'
```

A single quote can be included in a string by preceding it with a backslash, "\".

```
'It\'s easy to use ABEL'
```

is the string, *"It's easy to use ABEL"*.

Backslashes can be put in a string by using two of them in succession.

```
'He\\ she can use backslashes in a string'
```

becomes the string:

```
"He\ she can use backslashes in a string"
```

NOTE: Back-quotes (`) are also accepted as string delimiters and can be used interchangeably with the forward quote (').

COMMENTS 2.5

Comments are another way to make a source file easy to understand. Comments explain what is not readily apparent from the source code itself. Comments do not affect the meaning of the code.

A comment begins with a double quotation mark, ", and ends with either another double quotation mark or the end of line, whichever comes first. The text of the comment follows the opening quotation mark.

Comments cannot be imbedded within keywords.

Valid comments (shown in boldface):

```
MODULE Basic_Logic; "gives the module a name
TITLE 'ABEL design example: simple gates'; "title

"declaration section"
IC4 device 'P10L8'; "declare IC4 to be a P10L8
IC5 "decoder PAL" device 'P10H8';
```

Note that the information inside single quotation marks (apostrophes) is not a comment but is part of the statement.

NUMBERS 2.6

All operations in ABEL™ involving numeric values are done with 32-bit accuracy. Thus, valid numeric values fall in the range 0 to $2^{32} - 1$. Numbers are represented in any of five forms. The four most common forms represent numbers in different bases. The fifth form uses alphabetic characters to represent a numeric value.

When one of the four bases other than the default base is chosen to represent a number, the base used is indicated by a symbol preceding the number. Table 2-3 gives the four bases supported by ABEL™ and their accompanying symbols. The base symbols can be typed in upper-case or lower-case.

When a number is specified and is not preceded by a base symbol, it is assumed to be in the default base numbering system. The normal default base is base 10, so numbers are represented in decimal form unless preceded by a symbol indicating that another base is to be used.

For special applications, the default base can be changed. See "@ RADIX," in section 4, for more information.

Table 2-3. Number Representation in Different Bases

Base Name	Base	Symbol
binary	2	^b
octal	8	^o
decimal	10	^d
hexadecimal	16	^h

Here are some examples of valid number specifications. The default base is base ten.

Specification	Decimal Value
75	75
^h75	117
^b101	5
^o17	15
^h0F	15

The circumflex, "^", must be entered as a character from the keyboard. It does not represent a control key sequence as in some other popular software.

Numbers may also be specified by alphabetic characters. In this case, the numeric ASCII code of the letter is used as the numeric value. For example, the character "a" is decimal 97, hexadecimal 61 in the ASCII coding. The decimal value 97 would be used if "a" were specified as a number.

Sequences of alphabetic characters are first converted to their binary ASCII values, and then are concatenated to form numbers (usually large).

Some examples of numbers specified with characters are given here:

Specification	Hex Value	Decimal Value
'a'	^h61	97
'b'	^h62	98
'abc'	^h616263	6382203

SPECIAL CONSTANT VALUES 2.7

Constant, non-changing values can be used in ABEL™ logic descriptions. Constant values are used in assignment statements, truth tables and test vectors and are sometimes assigned to an identifier that then denotes that value throughout a module (see "Declarations," section 3.5, and "Module Statement," section 3.2).

Constant values may be either numeric or one of the non-numeric special constant values. The special values are listed in table 2-4.

Table 2-4. Special Constant Values

Value	Description
.C.	clocked input (low-high-low transition)
.F.	floating input or output signal
.K.	clocked input (high-low-high transition)
.P.	register preload
.SVn.	n = 2 through 9. Drive the input to super voltage 2 through 9.
.X.	don't care condition
.Z.	test input or output for high impedance

When one of the special constants is used, it must be entered as shown in table 2-4 with surrounding periods. The periods indicate that a special constant is being used; without the periods, .C. would appear to be an identifier named C. Special constants can be entered in either upper-case or lower-case.

OPERATORS, EXPRESSIONS, AND EQUATIONS 2.8

Items such as constants and signal names can be brought together in expressions. Expressions combine, compare or perform operations on the items they include to produce a single result. The operations to be performed (addition and logical AND are just two examples) are indicated by operators within the expression.

ABEL™ operators are divided into four basic types: logical, arithmetic, relational and assignment. Each of these types is discussed separately below, followed by a description of how they are combined into expressions, then by a summary of all the operators and the rules governing them, and finally by an explanation of how equations utilize expressions.

Logical Operators 2.8.1

Logical operators are used in Boolean expressions. ABEL™ incorporates the standard logical operators used in most logic designs; these operators are listed in table 2–5.

Logical operations involving operands of more than one bit are performed bit by bit. Thus, 2#4 equals 6. For alternate operators, refer to the @ALTERNATE Directive section in this Language Reference.

Table 2–5. Logical Operators

Operator	Example	Description
!	! A	NOT : ones complement
&	A & B	AND
#	A # B	OR
$	A $ B	XOR : exclusive OR
!$	A !$ B	XNOR : exclusive NOR

Arithmetic Operators 2.8.2

Arithmetic operators define arithmetic relationships between items in an expression. The shift operators are included in this class because each left shift of one bit is equivalent to multiplication by 2 and a right shift of one bit is the same as division by 2. Table 2–6 lists the arithmetic operators.

Table 2–6. Arithmetic Operators

Operator	Example	Description
-	-A	twos complement
-	A - B	subtraction
+	A + B	addition
*	A * B	multiplication
/	A / B	unsigned integer division
%	A % B	modulus: remainder from /
<<	A << B	shift A left by B bits
>>	A >> B	shift A right by B bits

Note that a minus sign has a different significance depending on its usage. When used with one operand, it indicates that the twos complement of the operand is to be formed. When the minus sign is found between two operands, the twos complements of the second operand is added to the first.

Division is unsigned integer division: the result of division is a positive integer. The remainder of a division can be obtained by using the modulus operator, "%".

The shift operators perform logical unsigned shifts. Zeroes are shifted in from the left during right shifts and in from the right during left shifts.

Relational Operators 2.8.3

Relational operators are used to compare two items in an expression. Expressions formed with relational operators produce a Boolean true or false value. Table 2-7 lists the relational operators.

Table 2-7. Relational Operators

Operator	Example	Description
==	A == B	equal
!=	A != B	not equal
<	A < B	less than
<=	A <= B	less than or equal
>	A > B	greater than
>=	A >= B	greater than or equal

All relational operations are unsigned. For example, the expression -1 > 4 is true since the twos complement of 1 is 1111, which is 15 in unsigned binary, and 15 is greater than 4. For the purpose of this example, a four-bit representation was assumed; in actual use, -1, the twos complement of 1, is 32 bits all set to 1.

Some examples of relational operators in expressions follow:

Expression	Value
2 == 3	false
2 != 3	true
3 < 5	true
-1 > 2	true

The logical values, true and false, are represented internally by numbers. Logical true is –1 in twos complement: all 32 bits are set to 1. Logical false is 0 in twos complement so all 32 bits are set to 0. This implies that an expression producing a true or false value (a relational expression) can be used anywhere a number or numeric expression could be used and –1 or 0 will be substituted in the expression depending on the logical result.

For example:

```
A = D $ (B == C) ;
```

means that:

A will equal the complement of D if B equals C

A will equal D if B does not equal C.

Assignment Operators 2.8.4

Assignment operators are a special class of operators used in equations rather than in expressions. Equations assign the value of an expression to output signals. See section 2.8.6 for a complete discussion of equations.

There are two assignment operators, unclocked and clocked. Unclocked or immediate assignment occurs without any delay as soon as the equation is evaluated. Clocked assignment occurs at the next clock pulse from the clock associated with the output. Table 2-8 shows the assignment operators.

Table 2-8. Assignment Operators

Operator	Description
=	Unclocked assignment (combinatorial outputs)
:=	Clocked assignment (registered outputs)

Expressions 2.8.5

Expressions are combinations of identifiers and operators that produce one result when evaluated. Any logical, arithmetic or relational operators may be used in expressions.

Expressions are evaluated according to the particular operators involved. Some operators take precedence over others, and their operation will be performed first. Each operator has been assigned a priority that determines the order of evaluation. Priority 1 is the highest priority, and priority 4 is the lowest.

Table 2–9 summarizes the logical, arithmetic and relational operators, presented in groups according to their priority.

When operations of the same priority exist in the same expression, they are performed in the order found from left to right in that expression. Parentheses may be used as in normal mathematics to change the order of evaluation, with the operation in the innermost set of parentheses performed first.

Some examples of valid expressions are given here. Note how the order of operations and the use of parentheses affect the evaluated result.

Expression	Result	Comments
2*3/2	3	operators with same priority
2 * 3 / 2	3	spaces are OK
2*(3/2)	2	fraction is truncated
2+3*4	14	
2#4$2	4	
2#(4$2)	6	
2==^HA	0	false
14==^HE	–1	true

Table 2-9. Summary of Operators and Priorities

Priority	Operator	Description
1	-	negate, twos complement
1	!	NOT, ones complement
2	&	AND
2	<<	shift left
2	>>	shift right
2	*	multiply
2	/	unsigned division
2	%	modulus, remainder from /
3	+	add
3	-	subtract
3	#	OR
3	$	XOR : exclusive OR
3	!$	XNOR : exclusive NOR
4	==	equal
4	!=	not equal
4	<	less than
4	<=	less than or equal
4	>	greater than
4	>=	greater than or equal

Equations 2.8.6

[!]... [ENABLE] element = expression ;

or

[!]... [ENABLE] element := expression ;

element	an identifier naming a signal or set of signals, or an actual set, to which the value of *expression* will be assigned
expression	any valid expression
= and :=	unclocked and clocked assignment operators

Equations assign the value of an expression to a signal or set of signals in a logic description. The identifier and expression must follow the rules already established for those elements.

Equations use the two assignment operators "=" (unclocked) and ":=" (clocked) described in section 2.8.4.

The keyword, *ENABLE*, is used to enable tri-state output buffers. The identifier must be a tri-state type output, and the value assignment applies only to the buffer enable rather than to the signal itself.

The complement operator, "!", can be used to express negative logic. The complement operator precedes the signal name and implies that the expression on the right of the equation is to be complemented before it is assigned to the signal. Use of the complement operator on the left side of equations is provided as an option; equations for negative logic parts can just as easily be expressed by complementing the expression on the right side of the equation.

Examples of equations:

```
X = A & B ;          " unclocked assignment to X
```

```
ENABLE Y = C # D ;     " Y is enabled if C or D is true

Y := A & B ;           " clocked assignment to Y

!A = B & C # D ;       " same as A = !(B & C # D);

!!A = B & C # D ;      " same as A = B & C # D ;
```

Multiple Assignments to the Same Identifier

When an identifier appears on the left side of more than one equation, the expressions being assigned to the identifier are first ORed together and then the assignment is made. If the identifier on the left side of the equation is complemented, the complement is performed after all the expressions have been ORed.

Examples:

Equations Found	Equivalent Equation
A = B; A = C;	A = B # C ;
A = B; A = C & D;	A = B # (C & D) ;
A = !B ; A = !C ;	A = !B # !C ;
!A = B ; !A = C ;	A = !(B # C) ;
!A = B ; A = !C ;	A = !C # !B ;
!A = B ; !A = C ; A = !D ; A = !E ;	A = !D # !E # !(B#C) ;

Note that when the complement operator appears on the left side of multiple assignment equations, the right-hand sides are ORed first and then the complement is applied.

SETS 2.9

A set is a collection of signals and constants that is operated on as one unit. Any operation applied to a set is applied to each element in the set. Sets simplify ABEL™ logic descriptions and test vectors by allowing groups of signals to be referenced with one name.

For example, the outputs B0–B7 of an eight-bit multiplexer could be collected into the set named *MULTOUT*. The three selection lines might be collected in the set, *SELECT*. The multiplexer could then be defined in terms of *MULTOUT* and *SELECT* rather than being defined by all the input and output bits individually specified.

A set is represented by a list of constants and signals separated by commas, or the range operator (..), and surrounded by square brackets. For example:

Sample Set	Description
[B0,B1,B2,B3,B4,B5,B6,B7]	outputs (*MULTOUT*)
[S0,S1,S2]	select lines (*SELECT*)

The above sets could also be expressed by using the range operator; for example:

```
[B0..B7]
[S0..S2]
```

Identifiers, used to delimit a range, must have compatible names; they must begin with the same alphabetical prefix and have a numerical suffix. Range identifiers can also delimit a decrementing range or a range which appears as one element of a larger set; for example:

[A7..A0]	decrementing range
[.X.,.X.,.X.,.X.,.X.,A10..A7]	range within a larger set

Note that for set specifications the square brackets **do not denote optional items**. The brackets are required to delimit the set.

Note also that ABEL™ sets are not mathematical sets.

Set Operations 2.9.1

Most operators can be applied to sets. In general, this means that the operation is performed on each element of the set, sometimes individually and sometimes according to the rules of Boolean algebra. Table 2-10 lists the operators that may be used with sets. Appendix C describes how these operators are applied to sets.

Table 2-10. Valid Set Operations

Operator	Example	Description
=	A = 5	assignment
:=	A := [1,0,1]	clocked assignment
!	!A	NOT : ones complement
&	A & B	AND
#	A # B	OR
$	A $ B	XOR : exclusive OR
!$	A !$ B	XNOR : exclusive NOR
-	-A	negate, twos complement
-	A - B	subtraction
+	A + B	addition
==	A == B	equal
!=	A != B	not equal
<	A < B	less than
<=	A <= B	less than or equal
>	A > B	greater than
>=	A >= B	greater than or equal

For operations involving two or more sets, the sets must have the same number of elements. The expression, "[*a*,*b*] + [*c*,*d*,*e*]", is invalid because the sets have different numbers of elements.

Some examples of set usage are given here.

The Boolean equation,

```
Chip_Sel = A15 & !A14 & A13 ;
```

represents an address decoder where *A15*, *A14* and *A13* are the three high-order bits of a 16-bit address. The decoder can easily be implemented with set operations.

First, a constant set that holds the address lines is defined so that the set can be referenced by name. This definition is done in the constant declaration section of a module (described in section 3). The declaration is:

```
Addr = [A15,A14,A13] ;
```

which declares the constant set *Addr*. The equation,

```
Chip_Sel = Addr == [1,0,1] ;
```

is functionally equivalent to:

```
Chip_Sel = A15 & !A14 & A13;
```

If *Addr* is equal to [1,0,1], meaning that *A15* = 1, *A14* = 0 and *A13* = 1, then *Chip_Sel* is set to true. Note that the set equation could also have been written as:

```
Chip_Sel = Addr == 5 ;
```

because 101 binary equals 5 decimal.

In the example above, a special set with the high-order bits of the 16-bit address was declared and used in the set operation. The full address could have been used and the same function arrived at in other ways, two of which are shown here:

Example 1:

```
" declare some constants in declaration section
Addr = [a15..a0] ;
X = .X. ; "simplify notation for don't care constant

Chip_Sel = Addr == [1,0,1,X,X,X,X,X,X,X,X,X,X,X,X] ;
```

Example 2:

```
" declare some constants in declaration section
Addr = [a15..a0] ;
X = .X. ;

Chip_Sel = ( Addr >= ^HA000 ) & ( Addr <=
                ^HBFFF ) ;
```

Both of the solutions presented in these final two examples are functionally equivalent to the original Boolean equation and to the first solution in which only the high order bits are specified as elements of the set (*Addr* = [*a15,a14,a13*]).

Set Assignment and Comparison 2.9.2

Values and sets of values can be assigned and compared to a set. For example,

```
sigset = [1,1,0] & [0,1,1] ;
```

results in *sigset* being assigned the value, [0,1,0].

Numbers in any representation can be assigned or compared to a set. The preceding set equation could have been written as

```
sigset = 6 & 3 ;
```

When numbers are used for set assignment or comparison, the number is converted to its binary representation and the following two rules apply:

1. If the number of significant bits in the binary representation of a number is greater than the number of elements in a set, the bits are truncated on the left.

2. If the number of significant bits in the binary representation of a number is less than the number of elements in a set, the number is padded on the left with leading zeroes.

Thus, the following two assignments are equivalent:

```
[a,b] = ^B101011 ;
[a,b] = ^B11 ;
```

And so are these two:

```
[d,c] = ^B01 ;
[d,c] = ^B1 ;
```

The set assignment,

```
[a,b] = c & d ;
```

is the same as the two assignments:

```
a = c & d ;
b = c & d ;
```

Restrictions on Sets 2.9.3

The following restriction applies to sets:

- Because set assignment is applied to all elements of a set, sets with mixed combinational and registered outputs cannot be used on the left side of an equation. Assignments are either clocked (combinational) or unclocked (registered) according to the assignment operator used (= or :=) and a clocked assignment to a combinational output or an unclocked assignment to a registered output constitutes an error and will be flagged by the language processor.

BLOCKS 2.10

Blocks are sections of ASCII text enclosed in braces, "{" and "}". Blocks are used in macros and directives. The text contained in a block can be all on one line or can span many lines. Some examples of blocks follow.

```
{ this is a block }

{
this is also a block, and it
spans more than one line.
}

{ A = B # C ;
D = [0,1] + [1,0] ;
}
```

Blocks can be nested within other blocks, as shown below, where the block "{ D = A }" is nested within a larger block:

```
{ A = B $ C ;
{ D = A ;}
E = C ;
}
```

Blocks and nesting of blocks can be useful in macros and when used with directives. (See "Macro Declarations," section 3.5.5 and "Directives," section 4.)

If either a right or left brace is needed as a character in a block but does not denote the start or end of a nested block, it is preceded by a backslash. Thus,

```
{ \{ \} }
```

is the block containing the characters " { } ", with the spaces included.

ARGUMENTS AND ARGUMENT SUBSTITUTION 2.11

Variable values can be used in macros, modules and directives. These values are called the arguments of the construct that uses them. In ABEL™, a distinction must be made between two types of arguments: actual and dummy. Their definitions are given here.

dummy argument	an identifier that is used to indicate where an actual argument is to be substituted in the macro, module, or directive.
actual argument	the argument (value) used in the macro, directive or module. The actual argument is substituted for the dummy argument. An actual argument can be any text, including identifiers, numbers, strings, operators, sets, or any other element of ABEL™.

Dummy arguments are specified in macro declarations and in the bodies of macros, modules and directives. The dummy argument is preceded by a question mark in the places where an actual argument is to be substituted. The question mark distinguishes the dummy arguments from other ABEL™ identifiers occurring in the source file.

Take for example, the following macro declaration (macros are discussed fully in section 3 and an example of usage is presented in the design example file MACRO.ABL).

```
OR_EM MACRO (a,b,c) { ?a # ?b # ?c };
```

This defines a macro named *OR_EM* that is the logical OR of three arguments. These arguments are represented in the definition of the macro by the dummy arguments, *a*, *b*, and *c*. In the body of the macro, which is surrounded by braces, the dummy arguments are preceded by question marks to indicate that an actual argument will be substituted.

Now, the equation,

```
D = OR__EM( x,y,z&1) ;
```

invokes the *OR_ EM* macro with the actual arguments, *x*, *y*, and *z&1*. This results in the equation,

```
D = x # y # z&1 ;
```

Spaces (blanks) are significant in actual arguments. Actual arguments are substituted exactly as they appear. Note that in the example above, the actual argument *z&1* contains no spaces in the equation referring to *OR__EM*, and that in the expanded equation the argument appears again without spaces. Had the argument been specified as *"z & 1"* (note the spaces), the resulting equation would have contained those spaces. For example, the equation,

```
D = OR__EM( x,y,z & 1)
```

results in the equation,

```
D = x # y # z & 1 ;
```

Argument substitution occurs before the source file is checked for syntactic or logical correctness. This means that the code is checked for correctness with the actual arguments in place. Thus, if an actual argument violates a syntactic or logical rule, the parser will detect and report the error.

In review:

- Dummy arguments are place holders for actual arguments.
- A question mark preceding the dummy argument indicates that an actual argument is to be substituted.
- Actual arguments replace dummy arguments before the source file is checked for correctness.
- Spaces are significant in actual arguments.

Further discussion and examples of argument usage are given in the ABEL™ Applications Guide, in the explanations of modules and macros in section 3, and in the section 4 description of directives.

STRUCTURE 3

This section describes the structure of a complete ABEL™ design description. The language elements described in section 2 are combined within the proper structures to define Boolean equations, state machines and truth tables. The device or devices being described must be indicated and initial declarations must be made. Definitions of expected output from simulation of the device may also be made.

BASIC STRUCTURE 3.1

ABEL™ source files can be broken into independent parts called modules. Each module contains one or more complete logic descriptions. At the simplest level, an input file to the ABEL™ language processor consists of only one module; at the most complex level, an indefinite number of modules may be combined into one source file and processed at the same time.

Every module consists of several different sections, each with its own unique function. These sections are:

- Declarations of devices, pins, nodes, constants, attributes, and macros
- Boolean logic equations
- Truth tables
- State diagrams
- Explicit fuse declarations
- Test vectors

Some or all of these parts of a module may exist for any given application. Declarations must always be made. Figure 3–1 shows the structure of an ABEL™ source file.

Because a source file is a collection of one or more modules, each with its own beginning and end, different source files can be combined to form complete system designs in one source file.

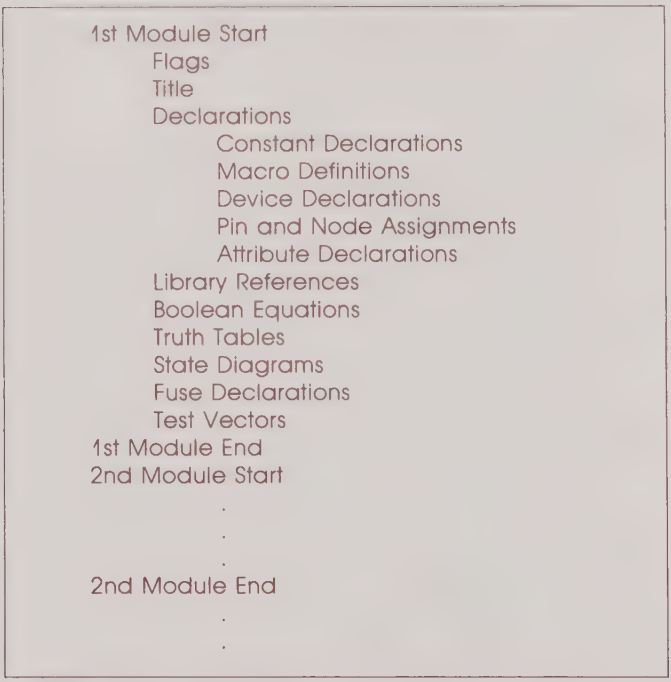


Figure 3–1. Structure of an ABEL™ Source File

MODULE STATEMENT AND STRUCTURE 3.2

```
MODULE modname [ ( dummy_arg [,dummy_arg]... ) ]
       [ FLAG statement ]...
       [ TITLE statement ]
       declarations
       [ EQUATIONS ]...
       [ TRUTH_TABLE ]...
       [ STATE_DIAGRAM ]...
       [ TEST_VECTORS ]...
END [ modname ] [ ; ]
```

modname	a valid identifier naming the module
dummy_arg	a dummy argument
FLAG	defines processing parameters to be passed to the language processor
TITLE	defines the title of the module
declarations	declarations section of the module in which pin, node, device, attribute, and constant declarations are made
EQUATIONS	Boolean logic equations section
TRUTH_TABLE	truth table specification
STATE_DIAGRAM	state machine specification
FUSES	explicit fuse declarations
TEST_VECTORS	specification of simulation test vectors

The module statement defines the beginning of a module and must be paired with an END statement that defines the module's end.

Each module in a source file should have a unique name.

The module contains the different declarations, equations, truth tables, state diagrams and simulation tables necessary to complete the logic description(s). The following three rules apply to module structure:

1. If the FLAG statement is used, it must be the first keyword used after the MODULE statement.

2. If the TITLE statement is used, it must be the first keyword used after the FLAG statement (if one exists). If FLAG is not used, the TITLE statement must be the first keyword used after the MODULE statement.

3. One declarations section must exist in a module. All other sections can occur in the module as many times as needed and in any order.

The optional dummy arguments in the module statement allow the actual arguments to be passed to the module when it is processed by the language processor. The dummy argument provides a name to refer to within the module. Anywhere in the module where a dummy argument is found preceded by a "?", the actual argument value will be substituted by the parser.

For example:

```
MODULE MY_EXAMPLE (A,B)
        .
        .
C = ?B + ?A
        .
        .
END
```

In the module named *MY_EXAMPLE*, C will take on the value of "*A* + *B*" where *A* and *B* contain actual arguments passed to the module when the language processor is invoked. For further information about dummy arguments, see section 2.11 in this manual.

FLAG STATEMENT 3.3

FLAG parameter[,parameter]...

parameter	a string containing valid command line parameters as defined in the ABEL™ User's Guide, excepting –I, –O, –N, and all PARSE parameters

The FLAG statement provides an alternative method of defining processing parameters that affect the way in which the source file is processed by the language processor. These parameters are normally passed from the command line or from a batch file when the language processor is invoked. FLAG allows the explicit statement of processing parameters directly in the source file. Parameters entered from the command line override parameters specified with the FLAG statement. Parse parameters, and the –I –O, and –N parameters are not allowed. Complete information about the command line parameters is contained in the ABEL™ User's Guide.

FLAG is useful when a source file requires a specific type or level of processing for that processing to be successful. For example, a design may be large or complex enough that it will generate too many product terms for the specified device unless the PRESTO reduction algorithm is used in the reduction step of the processing. You might also want to simulate the design at trace level 3. This is done with the following command:

```
flag '–r2','–t3'
```

TITLE STATEMENT 3.4

TITLE string

The title statement is used to give a module a title that will appear as a header in both the programmer load file and documentation file created by the language processor. The title is specified in the string following the keyword, TITLE.

Use of the title statement is optional.

If asterisks are found in the title string, they will not appear in the programmer load file header in order to conform with the JEDEC standard.

An example of a title statement that spans three lines and describes the logic design follows:

```
module m6809a
title '6809 memory decode
Jean Designer
Data I/O Corp Redmond WA 23 Jan 1984'
```

For more information concerning the programmer load file, refer to the ABEL™ User's Guide.

DECLARATIONS 3.5

The declarations section of a module specifies the device being described and the correspondence between the names used in a module and the pins and nodes of the device. Constants, attributes, and macros are also defined in the declarations section. Declarations stay in effect only in the module in which they are defined. Each module must have its own declarations section. There are six types of declaration statements:

Device	Node	Macro
Pin	Constant	Attribute

The syntax and use of each of these types is presented in the following subsections.

Device Declaration Statement 3.5.1

device_id [,device_id]... DEVICE real_device ;

device_id	an identifier used in a module for references to a device
real_device	a string describing the industry part number of the real device represented by *device_id*

The device declaration statement associates device names used in a module with actual programmable logic devices on which designs are implemented. The device name used in the logic description is specified with *device_id*. The industry part number of the programmable logic device is indicated by the string, *real_device*. A list of the devices currently supported by ABEL™ can be found in appendix D of the ABEL™ User's Guide. These are the only devices that can be specified by *real_device*.

The ending semicolon is required.

The following example specifies two device names, *u14* and *u15*, that represent F82S159 IFLs:

```
u14,u15  device  'F82S159' ;
```

Pin Declaration Statement 3.5.2

```
[!]pin__id [, [!]pin__id[... PIN [IN device__id]
          pin# [ ='attr[,attr]...'] [,pin# [ ='attr[,attr]...']]...
```

pin__id	an identifier used to refer to a pin throughout a module
device__id	a declared device name identifier indicating the device associated with these pins
pin#	the pin number on the real device
attr	a string that specifies pin attributes for devices with programmable pins. Valid strings are:

pos	positive polarity
neg	negative polarity
reg	registered signal
com	combinational signal
latch	latch input pin
feed__pin	feedback from pin
feed__reg	feedback from register
feed__or	feedback from OR-gate

The pin declaration statement indicates the correspondence between identifiers used in a module and the pins on a real device. The declaration can also specify pin attributes for devices with programmable pin characteristics.

When lists of *pin_ids* and *pin#*s are used in one pin declaration statement, there is a one-to-one correspondence between the identifiers and numbers given. There must be one pin number associated with each identifier listed.

When more than one device is declared in one module, the optional *IN* portion of the pin declaration must be used so that the pins are associated with the proper device. A device declaration must be made before the pin declarations. The ending semicolon is required. An example of a simple pin declaration follows:

```
!Clock,Reset,S1  PIN  IN  U12   12,15,3 ;
```

This pin declaration assigns the pin names, *Clock*, to pin 12 of device *U12*, *Reset* to pin 15, and *S1* to pin 3.

The use of the '!' operator in pin declarations indicates that the pin is active-low, and will be automatically negated when processed by the language processor.

The pin attribute, *attr*, specifies pin attributes. Attributes can be defined in this way or with the ISTYPE statement. The ISTYPE statement and attributes are discussed in section 3.5.6. The pin declaration,

```
F0 pin 13 ='neg, reg';
```

specifies that equations for *F0*, which corresponds to pin 13, should be optimized for a negative polarity registered output.

Node Declaration Statement 3.5.3

```
[!]node_id [,[!]node_id]... NODE [IN device_id] node#
        [ = 'attr[,attr]...'][,node# [ = 'attr[,attr]...']]...
```

node_id	an identifier used for reference to a node in a logic design
device_id	a declared device name identifier indicating the device associated with these nodes
node#	the node number on the real device
attr	a string that specifies node attributes for devices with programmable nodes. Valid strings are:

pos	positive polarity
neg	negative polarity
reg	registered signal
com	combinational signal
latch	latch input pin
feed_pin	feedback from pin
feed_reg	feedback from register
feed_or	feedback from OR-gate

The node declaration statement indicates the correspondence between identifiers used in a module and the internal nodes of a real device. Nodes are "pseudo-pins": internal signals that are not accessible on the device's external pins, but that are needed to program otherwise inaccessible fuses. The device nodes supported by ABEL™ are listed in appendix F of the ABEL™ User's Guide. The declaration can also specify node attributes for devices with programmable node characteristics.

When lists of *node_ids* and *node#s* are used in one node declaration statement, there is a one-to-one correspondence between the identifiers and numbers given. There must be one node number associated with each identifier listed.

When more than one device is declared in one module, the optional *IN* portion of the node declaration must be used so that the nodes are associated with the proper device.

A device declaration must be made before the node declarations.

The ending semicolon is required. The following example declares three nodes *A, B,* and C in the device *U15*.

```
A,B,C   NODE IN  U15   21,22,23 ;
```

The node attribute string, *attr*, specifies node attributes. Attributes can be defined in this way or with the ISTYPE statement. The ISTYPE statement and attributes are discussed in section 3.5.6.

The node declaration,

```
B   NODE 22  = 'pos,com';
```

specifies that node 22 has positive polarity and is a combinational node.

If shorthand notation is used to refer to internal device nodes, it is not necessary to declare the node in a declaration statement. For examples of shorthand notation of nodes, refer to section 5 in the Applications Guide in this manual. Appendix C in the Users Guide section lists the shorthand notation for all supported nodes.

Constant Declaration Statement 3.5.4

```
id [,id]... = expr [,expr]... ;
```

id	an identifier naming a constant to be used within a module
expr	an expression defining the constant value

The constant declaration statement defines constants to be used within a module. A constant is an identifier that retains a constant value throughout a module.

The identifiers listed on the left side of the equals sign in the constant declaration statement are assigned the values listed on the right side of the equals sign.

There is a one-to-one correspondence between the identifiers and the expressions listed and there must be one expression for each identifier.

The ending semicolon is required.

Constants are useful when a value must be used many times throughout a module and especially when the value is used many times but might need to be changed during the logic design process. Rather than changing the value throughout the module, the value can be changed once in the declaration of the constant.

Some examples of valid constant declarations follow:

```
ABC = 3*17;          " ABC is assigned the value 51
Y = 'Bc';            " Y = ^H4263 ;
X = .X.;             " X means 'don't care'
ADDR = [1,0,15];     " ADDR is a set with 3 elements
A,B,C = 5,[1,0],6;   " 3 constants declared here
D pin 6 ;            " see next line
E = [5*7,D];         " signal names can be included
G = [1,2]+[3,4];     " set operations are legal
A = B & C;           " operations on identifiers are
                     " valid
A = [!B,C];          " set and identifiers on right
```

Macro Declaration Statement and Macro Expansion 3.5.5

```
macro_id MACRO [ ( dummy_arg
                   [,dummy_arg]... ) ] block ;
```

macro_id	an identifier naming the macro
dummy_arg	a dummy argument
block	a block (see section 2)

The macro declaration statement defines a macro. Macros are used to include ABEL™ code in a source file without typing or copying the code everywhere it is needed. A macro is defined once in the declarations section of a module and then used anywhere within the module as frequently as needed. Macros can be used only within the module in which they are declared.

Wherever the *macro_id* occurs, the text in the block associated with that macro will be substituted. With the exception of dummy arguments, all text in the block (including spaces, end-of-lines, etc.) is substituted exactly as it appears in the block.

The dummy arguments used in the declaration of the macro allow different actual arguments to be used in the macro each time it is invoked in the module. Within the macro, dummy arguments are preceded by a "?" to indicate that an actual argument will be substituted for the dummy by the ABEL™ parser. This is best shown by example.

The equation,

```
NAND3 MACRO (A,B,C) { !(?A & ?B & ?C) } ;
```

declares a macro named *NAND3* with the dummy arguments *A*, *B* and C. The macro defines a three-input NAND gate. When the macro identifier occurs in the source, actual arguments for *A*, *B* and C will be supplied.

For example, the equation,

```
D = NAND3(Clock,Hello,Busy) ;
```

brings the text in the block associated with NAND3 into the code, with *Clock* substituted for *?A*, *Hello* for *?B* and *Busy* for ?C. This results in:

```
D = !(Clock & Hello & Busy) ;
```

which is the three-input NAND.

The macro *NAND3* has been specified by a Boolean equation, but it could have been specified using another ABEL™ construct, such as the truth table shown here:

```
NAND3 MACRO (A,B,C,Y)

    { TRUTH_TABLE ( [?A,?B,?C] -> ?Y )

    [ 0 ,.X.,.X.] -> 1 ;
    [.X., 0 ,.X.] -> 1 ;
    [.X.,.X., 0 ] -> 1 ;
    [ 1 , 1 , 1 ] -> 0 ; } ;
```

In this case, the line,

```
NAND3(Clock,Hello,Busy,D)
```

causes the text,

```
    TRUTH_TABLE ( [Clock,Hello,Busy] -> D )
    [ 0 ,.X.,.X.] -> 1 ;
    [.X., 0 ,.X.] -> 1 ;
    [.X.,.X., 0 ] -> 1 ;
    [ 1 , 1 , 1 ] -> 0 ;
```

to be substituted into the code. This text is a truth table definition of *D*, specified as the function of three inputs, *Clock*, *Hello* and *Busy*. This is the same function as that given by the Boolean equation, above. The truth table format is discussed in section 3.7.

Other examples of macros:

```
A macro { W = S1 & s2 & s3 ;} ;
"macro w/ no dummy args

B MACRO (d) { !?d } ; "macro w/ 1 dummy argument
```

and when they occur in logic descriptions:

```
A
X = W + B(inp) ;
Y = W + B( )C ;  "note the blank actual argument
```

resulting in:

```
"note leading space from block in A
  W = S1 & S2 & S3 ;
X = W + !inp ;
Y = W + ! C ;
```

ISTYPE Statement 3.5.6

```
signal [,signal]... ISTYPE [IN device_id] 'attr [,attr]...';

signal      a pin or node identifier

device_id   a declared device name identifier
            indicating the device associated with
            these nodes

attr        pos         positive polarity
            neg         negative polarity
            reg         registered signal
            com         combinational signal
            latch       latch input signal
            feed_pin    feedback from pin
            feed_reg    feedback from register
            feed_or     feedback from OR-gate
```

The ISTYPE statement defines attributes or characteristics of pins and nodes for devices with programmable characteristics. The attributes are used to form correct logic for the device and to optimize equations for the device. If no attributres are defined for a device with programmable characteristics the device defaults are applied. If attributes are defined for a device without programmable characteristics, an error is reported by the language processor. An example of ISTYPE statement usage is presented in the design example file ALTERA.ABL.

When more than one *signal* is listed on the left side of the ISTYPE statements, the attributes listed on the right side of the ISTYPE statement are applied to those signals.

When more than one device is declared in one module, the optional *IN DEVICE_id* portion of the ISTYPE must be used so the signals (pins or nodes) are associated with the correct device. A device declaration must be made before the ISTYPE statement appears in the source file. Declarations of the pin and node names used in the ISTYPE statement must be made before the ISTYPE statement.

An example of the ISTYPE statement follows:

```
FO, A  istype  'neg, latch' ;
```

This declaration statement defines *F0* and *A* as negative polarity latches. Both *F0* and *A* had to have been defined previously in the module.

Definitions of each of the attributes follows:

Pos (Positive Polarity)

Pos indicates that the associated input or output has positive polarity. The device will be programmed to reflect this condition and any equations associated with this signal will be optimized for that polarity. *Pos* may be entered in upper–, lower–, or mixed–case letters.

Neg (Negative Polarity)

Neg indicates that the associated input or output has negative polarity. The device will be programmed to reflect this condition and any equations associated with this signal will be optimized for negative polarity. *Neg* may be entered in upper–, lower–, or mixed–case letters.

Reg (Registered Signal)

Reg indicates that the associated input or output is registered rather than combinational. The device will be programmed for this condition, and the signal will change on application of the clock.

Com (Combinational Signal)

Com indicates that the associated input or output is a combinational or combinatorial signal. The device will be programmed for this condition.

Latch

The *latch* attribute indicates that the associated input is a latched input. *Latch* is valid only for inputs and causes an error if assigned to an output. Latches have an enable line and operate as follows:

1. If the enable is high, the input signal to the latch is passed through to the output.
2. If the enable is low, the last value present on the input line before the high–to–low transition of the enable is held on the output line.

Figure 3-2 shows a block diagram of a latch and a timing diagram for its operation.

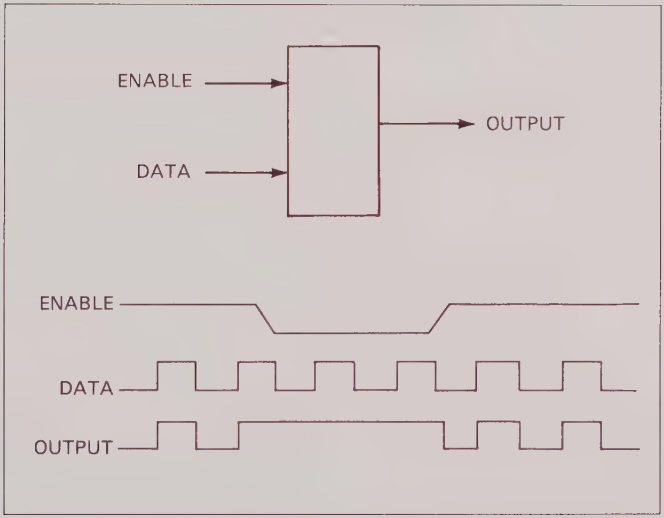
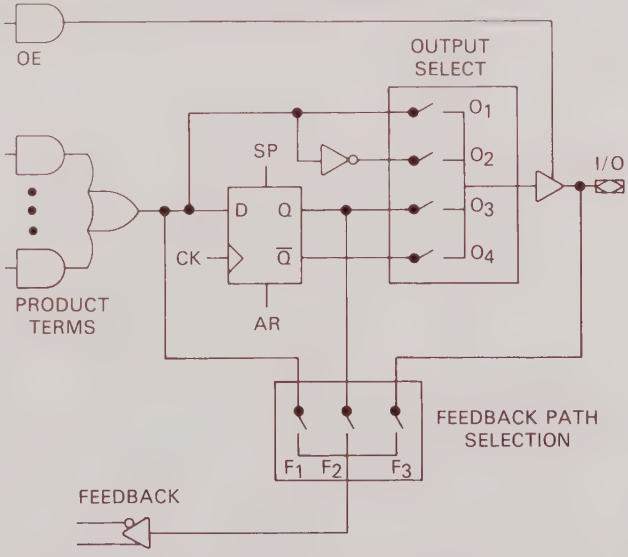


Figure 3–2. Block Diagram and Timing Diagram for a Latch

Feed_pin, Feed_reg, and Feed_or (Feedback Specifications)

Some available devices allow definition of the internal feedback paths. Feedback can occur from the output pin, the output of an internal register, or from the OR-gate output, as shown in figure 3–3.

The *feed_pin*, *feed_reg*, or *feed_or* attributes specify which feedback path to use: from the pin, the register output, or the OR-gate, respectively. Only one of the feedback attributes can be in effect at any given time; if more than one feedback attribute is specified for the same signal, the last attribute specified takes effect. Also, feedback attributes are valid only for devices with feedback paths that can be defined by the user. Feedback attributes are valid only for outputs. The attribute may be specified in upper–, lower–, or mixed–case characters.



F_1, feedback from OR-gate; selected by Feed_or
F_2, feedback from register; selected by Feed_reg
F_3, feedback from output pin; selected by Feed_pin

Figure 3–3. Feedback Paths for an EP300

EQUATIONS STATEMENT 3.6

EQUATIONS [IN device_id]

device_id	a previously declared device identifier that indicates the device associated with these equations

The equations statement defines the beginning of a group of equations associated with a device. Equations specify logic functions with Boolean algebra.

The equations statement is followed by the equations associated with the device indicated by the device identifier. In modules with only one device, the device identifier is optional; in all other cases it must be specified so that the equations will be applied toward the correct device.

The equations following the equation statement are any valid ABEL™ equations as described in section 2.8.6.

A sample equations section follows:

```
EQUATIONS IN IC13
    A = B & C # A ;
    [W,Y] = 3 ;
    !F = B==C ;
```

TRUTH TABLES 3.7

Truth tables are another way to describe logic designs with ABEL™ and may be used in lieu of, or in addition to, Boolean equations and state diagrams. Truth tables specify outputs as functions of different input combinations in a tabular form.

A truth table is specified with a header describing the format of the table and with the table itself.

Truth Table Header Syntax 3.7.1

```
TRUTH_TABLE [IN device_id] ( inputs -> outputs )

                    or

TRUTH_TABLE [IN device_id] ( inputs :> reg_outs )

                    or

TRUTH_TABLE [IN device_id] ( inputs :> reg_outs ->
                             outputs )
```

device_id	an identifier naming the device associated with the truth table
inputs	the inputs to the logic function
outputs	the outputs from the logic function
reg_outs	the registered (clocked) outputs
->	indicates the input to output function for combinational outputs.
:>	indicates the input to output function for registered outputs.

The truth table header can have one of the three forms shown above, depending on whether the device has registered or combinational outputs or both.

In all three forms, the device identifier is required when more than one device is declared in a module. An example of truth table usage is presented in the design example file LED7.ABL.

The inputs and outputs (both registered and combinational) of a truth table are either single signals, or, more frequently, sets of signals. If only one signal is used as either the input or output, its name is specified. Sets of signals used as inputs or outputs are specified in the normal set notation with the signals surrounded by square brackets and separated by commas (see section 2.9).

The syntax shown in the first form defines the format of a truth table with simple combinational outputs. The values of the inputs determine the values of the outputs.

The second form describes a format for a truth table with registered outputs. The symbol ":>" preceding the outputs distinguishes these outputs from the combinational ones. Again the values of the inputs determine the values of the outputs, but now the outputs are registered or clocked: they will contain the new value (as determined by the inputs) after the next clock pulse.

The third form is more complex, defining a table with both combinational and registered outputs. It is important in this format to make sure the different specification characters "->" and ":>" are used for the different types of outputs.

Examples of headers along with their accompanying truth tables are given after the format of the table is discussed.

Truth Table Format 3.7.2

The actual truth table (as opposed to the header) is specified according to the form described within the parentheses in the header. The truth table is a list of input combinations and resulting outputs. All or some of the possible input combinations may be listed.

As an example, the following truth table defines an exclusive-OR function with two inputs (*A* and *B*), one enable (*en*), and one output (C):

```
TRUTH_TABLE IN IC16 ( [en,A,B] -> C )

[0,.X.,.X.] -> .X.; " don't care w/ enable off
[1, 0 , 0 ] -> 0 ;
[1, 0 , 1 ] -> 1 ;
[1, 1 , 0 ] -> 1 ;
[1, 1 , 1 ] -> 0 ;
```

All values specified in the table must be constants, either declared, numeric, or the special constant, ".X.". Each line of the table (each input/output listing) must end with a semicolon.

Whereas the header defines the names of the inputs and outputs, the table defines the values of inputs and the resulting output values.

The following example shows a truth table description of a simple state machine with four states and one output. The current state is described by signals *A* and *B*, which are put into a set. The next state is described by the registered outputs *C* and *D*, which are also collected into a set. The single combinational output is signal *E*. The machine simply counts through the different states, driving the output *E* low when *A* equals 1 and *B* equals 0.

```
TRUTH_TABLE IN IC17 ( [A,B] :> [C,D] -> E )

0 :> 1 -> 1 ;
1 :> 2 -> 0 ;
2 :> 3 -> 1 ;
3 :> 0 -> 1 ;
```

Note that the input and output combinations are specified by a single constant value rather than by set notation. This is equivalent to:

```
[0,0] :> [0,1] -> 1 ;
[0,1] :> [1,0] -> 0 ;
[1,0] :> [1,1] -> 1 ;
[1,1] :> [0,0] -> 1 ;
```

STATE DIAGRAMS 3.8

An alternative to describing logic with Boolean equations and truth tables is to use a state diagram. The state diagram easily describes the operation of a sequential state machine implemented with programmable logic.

The specification of a state diagram requires the use of the STATE_DIAGRAM construct, which defines the state machine, and the IF-THEN-ELSE, CASE, and GOTO statements which determine the operation of the state machine.

The STATE_DIAGRAM construct is discussed first, and then the syntaxes of the IF-THEN-ELSE, CASE, and GOTO statements are presented.

Syntax of the WITH-ENDWITH statement, which permits the specification of equations in terms of transitions, is also presented.

STATE__DIAGRAM Statement 3.8.1

```
STATE__DIAGRAM [ IN device__id ] state__reg [ --> state__out ]
    [STATE state__exp : [equation]
                        [equation]
                            .
                            .
                            .
                        trans__stmt] . . .
```

device__id	an identifier specifying the associated device
state__reg	an identifier or set of identifiers specifying the signals that determine the current state of the machine.
state__out	an identifier or set of identifiers that determine the next state of the machine (for designs with external registers)
state__exp	an expression giving the current state
equation	a valid equation that defines the state machine outputs
trans__stmt	an IF-THEN-ELSE, CASE or GOTO statement, optionally followed by WITH__ENDWITH transition equations.

The STATE__DIAGRAM construct defines a state machine associated with a device in a module. The state machine starts in one of the states indicated by *state__exp*. The equations listed after that *state__exp* are evaluated, and the transition statement (*trans__stmt*) is evaluated after the next clock, causing the machine to advance to the next state.

Equations associated with a state are optional. Each state must have a transition statement. If none of the transition conditions for a state is met, the next state is undefined. (For some devices, undefined state transitions cause a transition to the cleared register state.) As many states as are needed can be specified.

When there is more than one device declared in a module, the *"IN device_id"* section is required.

Following is an example of a simple state machine that advances from one state to the next, setting the output to the current state, and then starting over again. Note that the states do not need to be specified in any particular order. Note also that state 2 is identified by an expression rather than by a constant. The state register is composed of the signals *a* and *b*.

```
state_diagram in u15 [a,b]
      state 3     : y = 3 ;
                    goto 0 ;
      state 1     : y = 1 ;
                    goto 2 ;
      state 0     : y = 0 ;
                    goto 1 ;
      state 1 + 1 : y = 2 ;
                    goto 3 ;
```

The next state diagram specifies a more complex state machine where the state_reg is specified with a constant set containing the signals *a* and *b*. Assuming that the state machine starts in state 1 (*a* = 0,b = 1), the sequence of states will be 1,4,2,3,2,4, 1,4,2,3,2,4, 1...

```
current_state = [a,b] "constant declaration

STATE_DIAGRAM current_state
        state 1: w = 1 ;
                 y = 1 ;
                 GOTO 4;
        state 2: IF y ==3 THEN 3
                          ELSE 4;
        state 3: w = 2;
                 y = w;
                 GOTO 2;
        state 4: y = 3;
                 CASE     w ==1: 2;
                          w ==2: 1;
                 ENDCASE;
```

Figure 3–4 shows the pictorial state diagram for this same state machine.

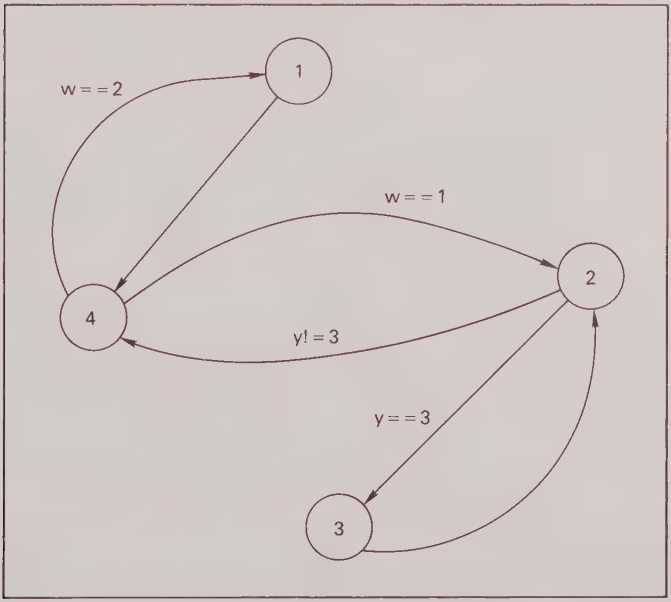


Figure 3–4. Pictorial State Diagram

IF–THEN–ELSE Statement 3.8.2

IF expression THEN state__exp [ELSE state__exp];

expression	any valid expression
state__exp	an expression identifying the next state, optionally followed by WITH__ENDWITH transition equations.

The IF–THEN–ELSE statement is an easy way to describe the progression from one state to another in a state machine. The expression following the IF keyword is evaluated, and if the result is true, the machine goes to the state indicated by the state__exp following the THEN keyword. If the result of the expression is false, the machine advances to the state indicated by the ELSE keyword.

Any number of IF statements may be used in a given state, and the use of the ELSE clause is optional.

Examples:

```
if A==B then 2 ;  "if A equals B goto state 2
if x-y then j else k ;  "if x-y is not 0 goto j, else goto k
if A then b*c ;  "if A is true (non-zero) goto state b*c
```

Chained IF–THEN–ELSE Statements

```
IF expression THEN state__expression
ELSE
  IF expression THEN state__expression
  ELSE
    IF expression THEN state__expression
    ELSE state__expression ;
```

Additional IF–THEN–ELSE statements can be chained to the ELSE clause of an IF–THEN–ELSE statement. Any number of IF–THEN–ELSE statements can be chained, but the final statement must end with a semicolon. An example of chained IF–THEN–ELSE statements follows:

```
if a then 1
else
  if b then 2
  else
    if c then 3
    else 0 ;
```

Often, chains of mutually exclusive IF–THEN–ELSE statements can be more clearly expressed with a CASE statement. The chained IF–THEN–ELSE statement is intended for situations (such as the preceding example) where the conditions are not mutually exclusive.

See section 3.8.1 for examples of IF–THEN–ELSE usage as they are used in a state diagram.

CASE Statement 3.8.3

```
CASE [ expression : state__exp; ]
     [ expression : state__exp; ]
     [ expression : state__exp; ]
          .
          .
          .
ENDCASE ;
```

expression	any valid ABEL™ expression
state__exp	an expression identifying the next state, optionally followed by WITH–ENDWITH transition equations.

The CASE statement is an easy way to indicate the transitions of a state machine when there are multiple possible conditions that affect the state transitions.

The expressions contained within the CASE–ENDCASE keywords must be mutually exclusive, meaning that only one of the expressions can be true at any given time. If two or more expressions within the same CASE statement are true, the resulting equations are undefined.

The state machine will advance to the state indicated by the *state__exp* following the expression that produces a true value. If no expression is true, the result is undefined, and the resulting action depends on the device being used. (For devices with D flip–flops, the next state will be the cleared register state.) For this reason, you should be sure to cover all possible conditions in the CASE statement expressions. If the expression produces a numeric rather than a logical value, 0 is false, and any non–zero value is true.

Example:

```
case a==0 : 1 ;
     a==1 : 2 ;
     a==2 : 3 ;
     a==3 : 0 ;
endcase ;
```

More CASE statement examples are presented in section 3.8.1.

GOTO Statement 3.8.4

GOTO state_exp ;

state_exp	an expression identifying the next state, optionally followed by WITH_ENDWITH transition equations.

The GOTO statement causes an unconditional transition to the state indicated by state_exp.

Example:

```
GOTO 0 ;      "goto state 0
GOTO x+y ;    "goto the state x+y
```

WITH-ENDWITH Statement 3.8.5

transition_ stmt	state__exp WITH equation [equation] . . ENDWITH ;
transition__stmt	IF, ELSE, or CASE statement
state__exp	the next state
equation	an equation for state machine outputs

The WITH-ENDWITH statement, when used in conjunction with the IF-THEN or CASE statement, allows output equations to be written in terms of transitions; for example:

```
state 5: IF a==1 then 1
                WITH x:=1;
                     y:=0;
                ENDWITH
         ELSE 2 WITH
                     x:=0;
                     y:=1;
                ENDWITH;
```

FUSES SECTION 3.9

```
FUSES [IN device_id]

fuse_number = fuse value;

    or

fuse_number_set = fuse value;
```

device_id	identifier specifying associated device
fuse_number	fuse number obtained form logic diagram of device
fuse_number_set	set of fuse numbers contained in square brackets
fuse_value	number indicating state of fuse(s)

The FUSES section of the source file provides a means for explicitly declaring the state of any fuse in the associated device. For example:

```
FUSES
  3552 = 1;
  [3478...3491]= ^Hff;
```

Fuse values appearing on the right side of the = symbol can be any number. In the case of only a single fuse number being specified on the left side of the = symbol, the least significant (LSB) bit of the fuse value is assigned to the fuse; a 1 indicates a fuse intact, and a 0 indicates a fuse blown. In the case of multiple fuse numbers, the fuse value is expanded to a binary number and truncated or padded to the left with zeros to obtain fuse values for each fuse number.

When fuse states are specified using the FUSES–section syntax, the resulting fuse values supercede the fuse values obtained through the use of equations, truth tables and state diagrams, and will effect device simulation accordingly.

TEST VECTORS 3.10

Test vectors specify the expected functional operation of a logic device by explicitly defining the device outputs as functions of the inputs. Test vectors are used for simulation of an internal model of the device and functional testing of the real programmed device.

A special simulation utility, SIMULATE, is provided as part of the ABEL™ software package. SIMULATE simulates the operation of the device model by applying the inputs specified in the test vectors to the fuse states created by the language processor. SIMULATE is discussed further in the ABEL™ User's Guide.

Functional testing of the real device is performed by a logic programmer after a device has been programmed. This can be done because the test vectors become part of the programmer load file that is loaded into the logic programmer.

Test vectors are written for each different device in the module, so that the different characteristics of each device can be taken into account separately during simulation.

Test vectors for a device are specified in a table. The table consists of a header and the vectors themselves. The header indicates that test vectors are to follow and defines the format of the table. The vectors specify the input-to-output function.

Test Vector Table Format 3.10.1

```
TEST_VECTORS [IN device_id] [note] ( inputs -> outputs )
[invalues -> outvalues ;]
.
.
.
```

device_id	an identifier indicating the device associated with this table
note	a string used to describe the test vectors
inputs	identifier or set of identifiers specifying the names of the input signals to the device, or feedback output signals
outputs	identifier or set of identifiers specifying the output signals from the device
invalues	input value or set of input values
outvalues	output value or set of output values resulting from the given inputs

The form of the test vectors is determined by the header. Each vector is specified in the format described within the parentheses in the header statement. An optional note string can be specified in the header. This note string is often used to describe what the vectors test, and is included as output in the simulation output file, the document output file, and the JEDEC programmer load file.

The table lists input combinations and their resulting outputs. All or some of the possible input combinations can be listed.

All values specified in the table must be constants, either declared, numeric or the special constant, ".X.". Each line of the table (each input/output listing) must end with a semicolon.

Following is a simple test vectors table:

```
TEST_VECTORS
( [A,B] -> [C,D] )

  [0,0] -> [1,1] ;
  [0,1] -> [1,0] ;
  [1,0] -> [0,1] ;
  [1,1] -> [0,0] ;
```

The following test vector table is equivalent to the table specified above because values for sets can be specified with numeric constants.

```
TEST_VECTORS
( [A,B] -> [C,D] )

  0 -> 3 ;
  1 -> 2 ;
  2 -> 1 ;
  3 -> 0 ;
```

If the signal identifiers used in the test vector header were declared as active-low in the declaration section, then constant values specified in the test vectors will be inverted accordingly.

DIRECTIVES 4

Directives provide many options that can affect the contents of a source file when it is processed. Sections of ABEL™ source code can be included conditionally, code can be brought into a source file from another file, and messages can be printed during the processing of a source file.

Directives are for the designer who understands the basics of ABEL™ and who wants to use more complex structures. Table 4–1 lists the available directives, which are then described in detail in the following subsections. Examples of directive usage to create test vectors are given in section 4.5 of the ABEL™ Applications Guide.

Some of the directives take arguments that are used to determine conditions. When this is the case, the argument can be an actual argument, or it can be a dummy argument preceded by a question mark.

The rules applying to actual and dummy arguments are presented in section 2.11.

Table 4–1. Directives

@ALTERNATE	@IFDEF	@INCLUDE	@RADIX
@CONST	@IFIDEN	@IRP	@REPEAT
@EXPR	@IFNB	@IRPC	@STANDARD
@EXIT	@IFNDEF	@MESSAGE	
@IF	@IFNIDEN	@PAGE	
@IFB			

@ALTERNATE

@ALTERNATE brings an alternate set of operators into effect that replace the normal ABEL™ operators. This is for users who feel more comfortable with the alternate set because of their familiarity with operators used in other languages.

The alternate operators remain in effect until the @STANDARD directive is issued or the end of the module is reached.

The alternate operator set is listed in table 4–2.

Table 4–2. Alternate Operator Set

ABEL™ Operator	Alternate Operator	Description
!	/	NOT
&	*	AND
#	+	OR
$	:+:	XOR
!$	:*:	XNOR

Note that the use of the alternate operator set precludes use of the ABEL™ addition, multiplication and division operators because they represent the OR, AND and NOT logical operators in the alternate set.

@CONST (Constant) Directive 4.2

```
@CONST id = expression ;

id          a valid identifier

expression  a valid expression
```

@CONST allows new constant declarations to be made in a source file outside the normal (and required) declarations section.

The @CONST directive is intended to be used inside macros so that they can define their own internal constants. Constants defined with @CONST override any previous constant declarations. Declaring an identifier as a constant in this manner constitutes an error if the identifier was used earlier in the source file as something other than a constant (i.e., a macro, pin, device).

Example:

```
@CONST count = count + 1;
```

@EXIT Directive 4.3

```
@EXIT
```

The @EXIT directive causes PARSE to abort processing of the source file with error bits set. (Error bits allow the operating system to determine that a processing error has occurred.)

@EXPR (Expression) Directive 4.4

@EXPR [block] expression ;

block	a block
expression	a valid expression

@EXPR evaluates the given expression, and converts it to a string of digits in the default base numbering system. This string and the block are then inserted into the source file at the point at which the @EXPR directive occurs. The expression must produce a valid number.

Example:

```
@expr {ABC} ^B11 ;
```

Assuming that the default base is base ten, this example causes the text *ABC3* to be inserted into the source file.

@IF Directive 4.5

@IF expression block

expression	a valid expression that produces a numeric value
block	a valid block of text as described in section 2

@IF is used to include sections of ABEL™ source code based on the value resulting from an expression. If the expression is non-zero (logical true), the block of code is included as part of the source.

Dummy argument substitution is valid in the expression.

Example:

```
@IF (A > 17) { C = D $ F ; }
```

@IFB (If Blank) Directive 4.6

@IFB (arg) block

arg	either an actual argument or a dummy argument preceded by a "?"
block	a valid block of text

@IFB includes the text contained within the block if the argument is blank (has 0 characters).

Examples:

```
@IFB ()
{ text here will be included
with the rest of the source file.
}

@IFB ( hello )
{ this text will not be included }

@IFB (?A)
{ this text will be included if no value is substituted
for A. }
```

@IFDEF (If Defined) Directive 4.7

@IFDEF id block

id	an identifier
block	a valid block of text

@IFDEF includes the text contained within the block if the identifier is defined.

Examples:

```
A pin 5 ;
@ifdef A { Base = ^hE000 ; }
"the above assignment is made because A was
"defined
```

@IFIDEN (If Identical) Directive 4.8

@IFIDEN (arg1,arg2) block

arg1,2	either an actual argument, or a dummy argument name preceded by a "?"
block	a valid block of text

The text in the block is included in the source file if arg1 and arg2 are identical.

Example:

```
@ifiden (?A,abcd) { ?A device 'P16R4'; }
```

A device declaration for a P16R4 is made if the actual argument substituted for *A* is identical to *abcd*.

@IFNB (If Not Blank) Directive 4.9

@IFNB (arg) block

arg	arg either an actual argument, or a dummy argument name preceded by a "?"
block	a valid block of text

@IFNB includes the text contained within the block if the argument is not blank, meaning that it has more than 0 characters.

Examples:

```
@IFNB ()
{ ABEL™ source here will not be included
with the rest of the source file.
}

@IFNB ( hello )
{ this text will be included }

@IFNB (?A)
{ this text will be included if a value is substituted for A. }
```

@IFNDEF (If Not Defined) Directive 4.10

@IFNDEF id block

id an identifier

block a valid block of text

@IFNDEF includes the text contained within the block if the identifier is undefined. Thus, if no declaration (pin, node, device, macro or constant) has been made for the identifier, the text in the block will be inserted into the source file.

Example:

```
@ifndef A{Base = ^hE000;}
"if A is not defined, the block is inserted in the text
```

@IFNIDEN (If Not Identical) Directive 4.11

@IFNIDEN (arg1,arg2) block

arg1,2 either an actual argument, or a dummy argument name preceded by a "?"

block a valid block of text

The text in the block is included in the source file if arg1 and arg2 are not identical.

Example:

```
@ifniden (?A,abcd) { ?A device 'P16R8'; }
```

A device declaration for a P16R8 is made if the actual argument substituted for *A* is not identical to *abcd*.

@INCLUDE Directive 4.12

@INCLUDE filespec

filespec	a string specifying the name of a file, where the specification follows the rules of the operating system being used

@INCLUDE causes the contents of the file identified by the file specification to be placed in the ABEL™ source file. The inclusion will begin at the location of the @INCLUDE directive. The file specification can include an explicit drive or path specification that indicates where the file is to be found. If no drive or path specification is given, the file is expected to be on either the default drive or path, or on the drive or path specified by the -H parameter (see section 4.2 of the ABEL™ User's Guide).

Example:

```
@INCLUDE 'macros.abl' '' file specification
```

@IRP (Indefinite Repeat) Directive 4.13

```
@IRP dummy_arg ( arg [,arg]... ) block
```

dummy_arg	a dummy argument
arg	either an actual argument, or a dummy argument name preceded by a "?"
block	a block of text

@IRP causes the block to be repeated in the source file *n* times, where *n* equals the number of arguments contained in the parentheses. Each time the block is repeated, the dummy argument takes on the value of the next successive argument.

For example:

```
@IRP A (1,^H0A,0)
{B = ?A ;
}
```

results in:

```
B = 1 ;
B = ^H0A ;
B = 0 ;
```

which is inserted into the source file at the location of the @IRP directive. Multiple assignments to the same identifier cause an implicit OR to occur (see section 2.8.6).

Note that if the directive is specified like this:

```
@IRP A (1,^H0A,0)
{B = ?A ; }
```

the resulting text would be:

```
B = 1 ; B = ^H0A ; B = 0 ;
```

The text appears all on one line because the block in the @IRP definition contains no end-of-lines. Remember that end-of-lines and spaces are significant in blocks.

@IRPC (Indefinite Repeat, Character) Directive 4.14

@IRPC dummy__arg (arg) block

dummy__arg	a dummy argument
arg	either an actual argument, or a dummy argument name preceded by a "?"
block	a block

@IRPC causes the block to be repeated in the source file *n* times, where *n* equals the number of characters contained in *arg*. Each time the block is repeated, the dummy argument takes on the value of the next successive character.

For example:

```
@IRPC A (Cat)
{B = ?A ;
}
```

results in:

```
B = C ;
B = a ;
B = t ;
```

which is inserted into the source file at the location of the @IRPC directive.

@MESSAGE Directive 4.15

```
@MESSAGE string

string    any valid string
```

@MESSAGE prints a message specified in *string* at the terminal. This can be used to monitor the progress of the PARSE step of the language processor.

Example:

```
@message 'Includes completed'
```

@PAGE Directive 4.16

```
@PAGE
```

Send a form feed to the parser listing file. If no listing is being created, @PAGE has no effect.

@RADIX Directive 4.17

```
@RADIX expr ;
```

expr a valid expression that produces the number 2, 8, 10 or 16 to indicate a new default base numbering.

@RADIX is used to change the default base numbering system. The normal default is base 10 (decimal). This directive is useful when many numbers need to be specified in a base other than 10, say base 2. The @RADIX directive can be issued and all numbers that do not have their base explicitly stated are assumed to be in the new base, in this case, base 2. (See section 2.6.)

The newly specified default base stays in effect until another @RADIX directive is issued or until the end of the module is reached.

When the default base is set to 16, all numbers in that base that begin with an alphabetic character must begin with leading zeroes.

Examples:

```
@radix 2 ;        "change default base to binary
@radix 11011/11 + 1 ;        "change back to decimal
```

@REPEAT Directive 4.18

@REPEAT expr block

expr a valid expression that produces a number

block a block

@REPEAT causes the block to be repeated *n* times, where *n* is specified by the constant expression.

The following use of the repeat directive,

```
@repeat 5 {H,}
```

results in the text *"H,H,H,H,H,"* being inserted into the source file. The @REPEAT directive is useful in generating long truth tables and sets of test vectors. Examples of @REPEAT usage can be found in the ABEL™ Applications Guide.

@STANDARD Directive 4.19

@STANDARD

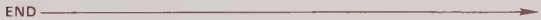
@STANDARD switches the operators in effect back to the ABEL™ standard operators from the alternate set. The alternate set is chosen with the @ALTERNATE directive.

Syntax Diagrams A

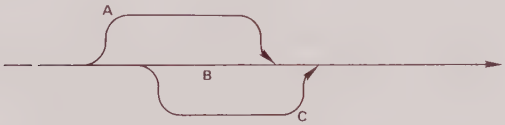
Following are syntax diagrams (sometimes called spaghetti diagrams) that formally and completely define the ABEL™ design language. Section A.1 teaches you how to read the syntax diagrams. The diagrams describing ABEL™ begin in section A.2.

How to Read Syntax Diagrams A.1

Each diagram is preceded on the left side by the name of the language element defined. Follow the diagram in the direction of the arrows to the far right where the diagram ends with a right arrow. In general, the flow is from left to right and top to bottom. The following diagram defines the END keyword.



Where an intersection exists with a choice of two or more paths, you may follow any one of them. In the example below, you can follow path *A*, *B*, or *C*.

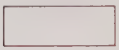


When a path diverges from the main line path and loops back on it, you can follow this loop as many times as desired. For example,

indicates that you can circle around loop *D* as many times as needed before proceeding along the straight line path *E* to the right.

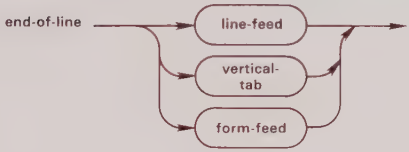
Certain shapes in the diagrams are significant. They are:

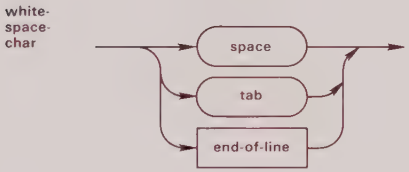
Oblong shapes and circles indicate keywords and punctuation that must be entered as shown.

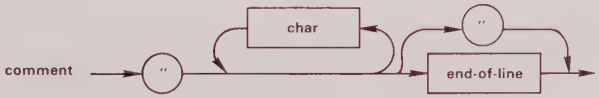


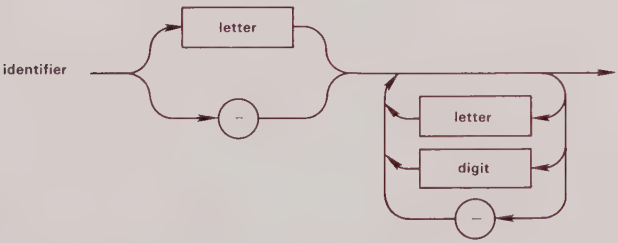
Rectangles indicate elements that are defined by other diagrams.

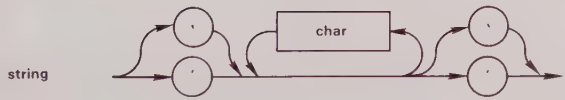
ABEL™ Syntax Diagrams A.2

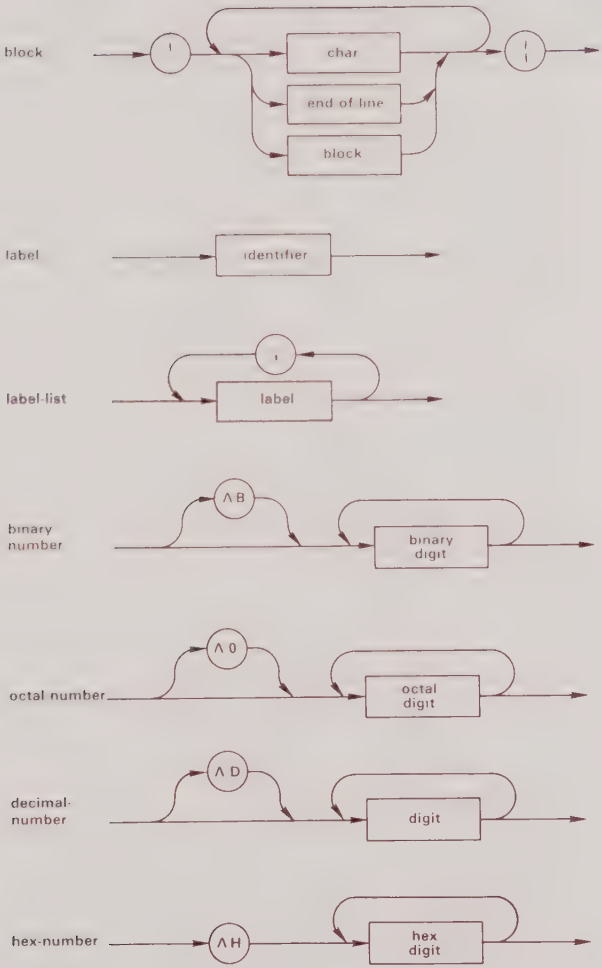


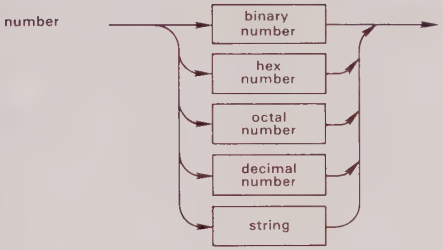


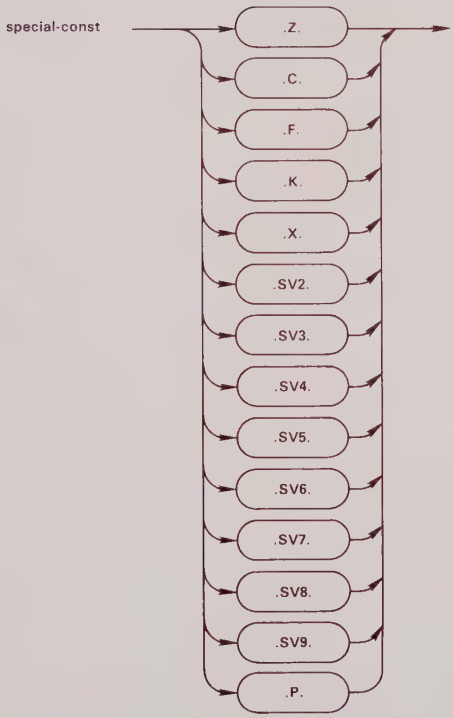


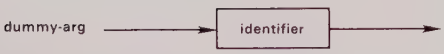




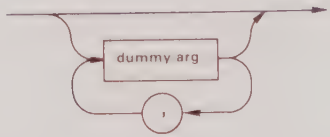

block
'
char
end of line
block
'
label
identifier
label-list
,
label
binary number
^B
binary digit
octal number
^O
octal digit
decimal-number
^D
digit
hex-number
^H
hex digit


number
binary number
hex number
octal number
decimal number
string

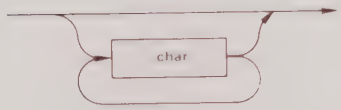

special-const
.Z.
.C.
.F.
.K.
.X.
.SV2.
.SV3.
.SV4.
.SV5.
.SV6.
.SV7.
.SV8.
.SV9.
.P.


dummy-arg
identifier

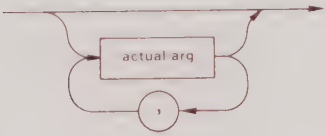
dummy arg list



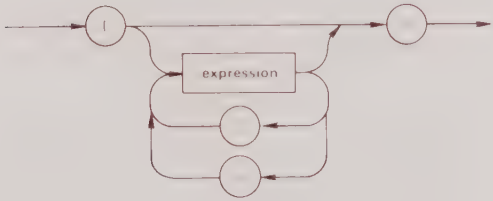
actual arg



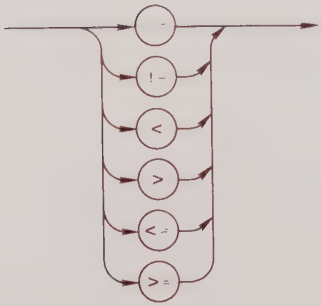
actual arg list



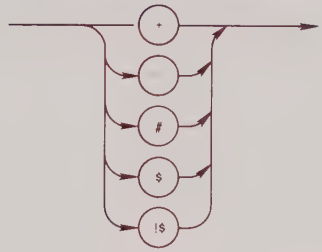
set



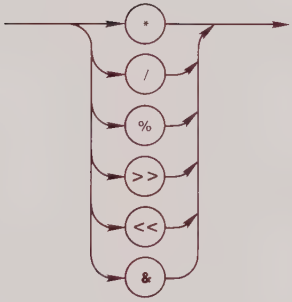
rel-op



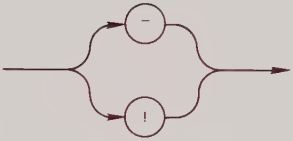
add-opp



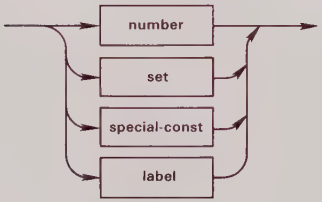
mul-op

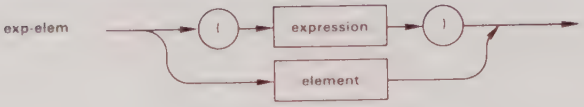


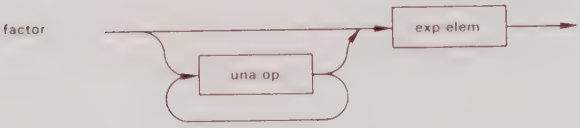
una-op

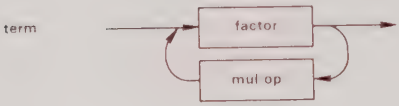


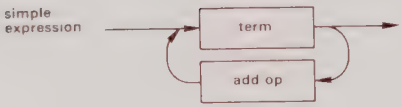
element

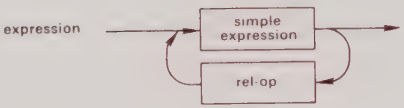


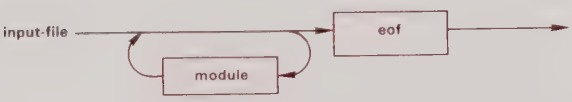

exp-elem
(
expression
)
element


factor
exp-elem
una-op

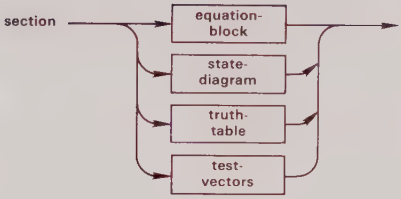

term
factor
mul-op

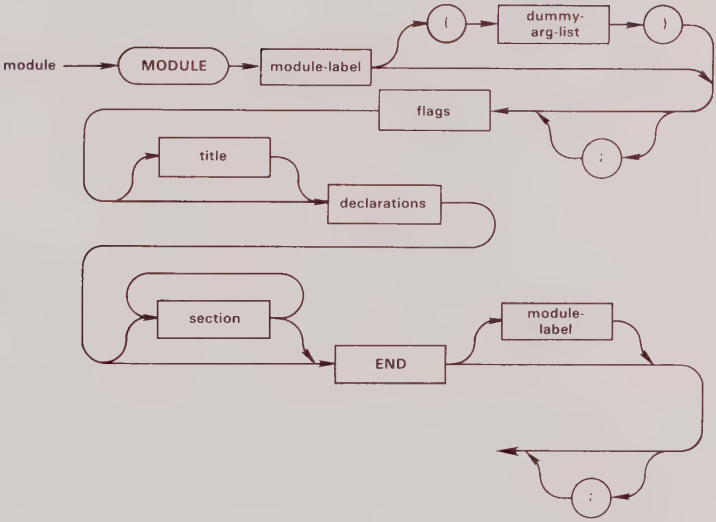

simple
expression
term
add op

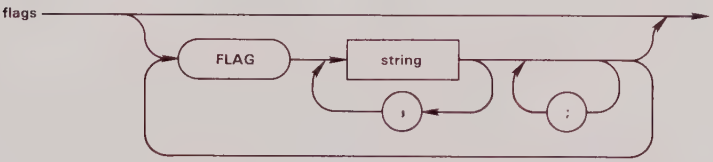

expression
simple
expression
rel-op

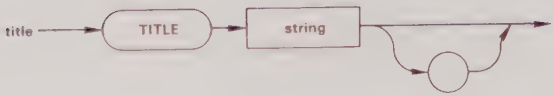

input-file
eof
module

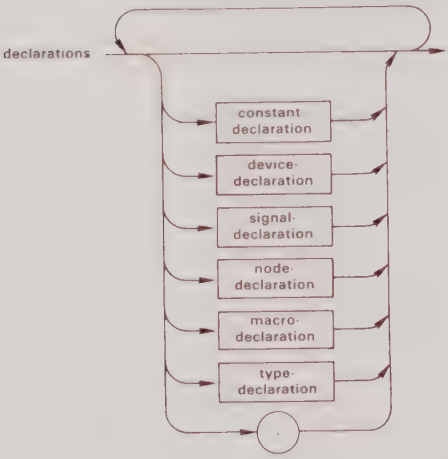
module-label
label

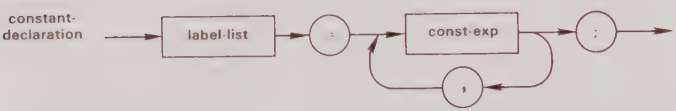

section
equation-block
state-diagram
truth-table
test-vectors


module
MODULE
module-label
(
dummy-arg-list
)
flags
;
title
declarations
section
END
module-label
;


flags
FLAG
string
,
;

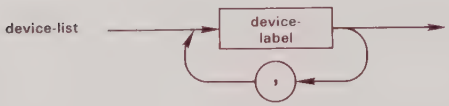

title
TITLE
string


declarations
constant-declaration
device-declaration
signal-declaration
node-declaration
macro-declaration
type-declaration
.

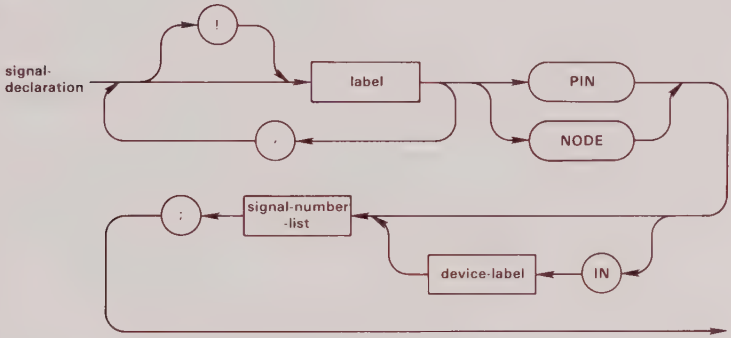

constant-declaration
label-list
:
const-exp
;
,

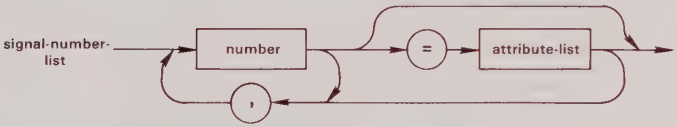
const-exp
expression

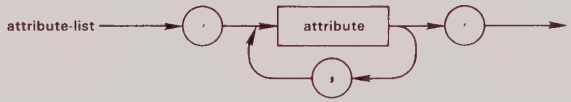
device-label
label

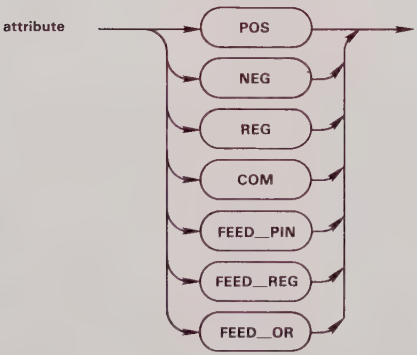

device-list
device-label
,

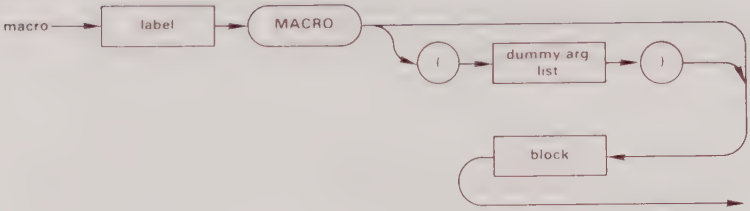
device declaration
device list
DEVICE
string
;

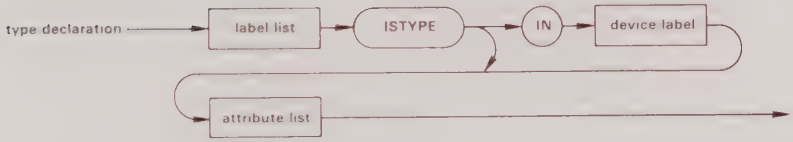

signal-declaration
!
,
label
PIN
NODE
;
signal-number-list
device-label
IN

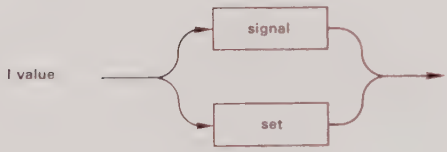

signal-number-list
number
,
=
attribute-list

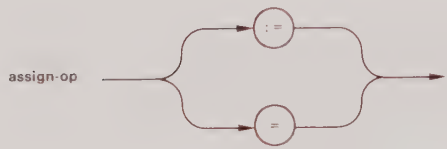

attribute-list
'
attribute
,
'

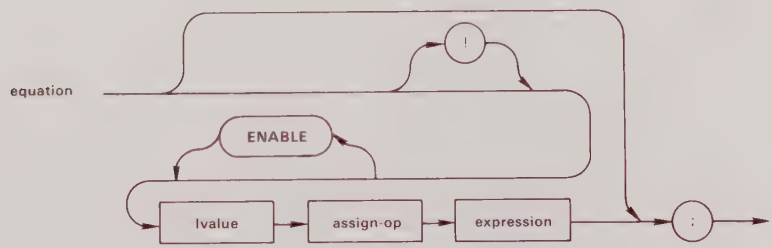

attribute
POS
NEG
REG
COM
FEED_PIN
FEED_REG
FEED_OR

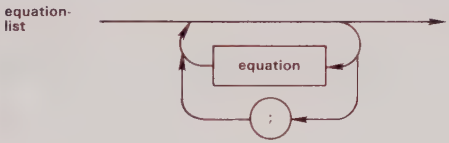

macro
label
MACRO
(
dummy arg list
)
block

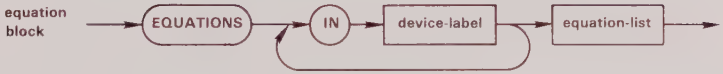

type declaration
label list
ISTYPE
IN
device label
attribute list

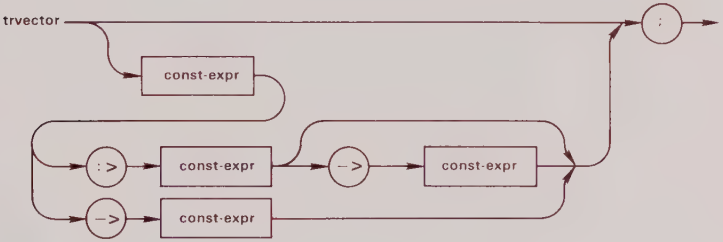

l value
signal
set

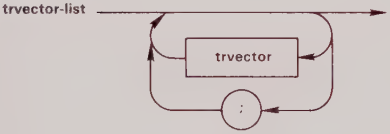

assign-op
: =
=

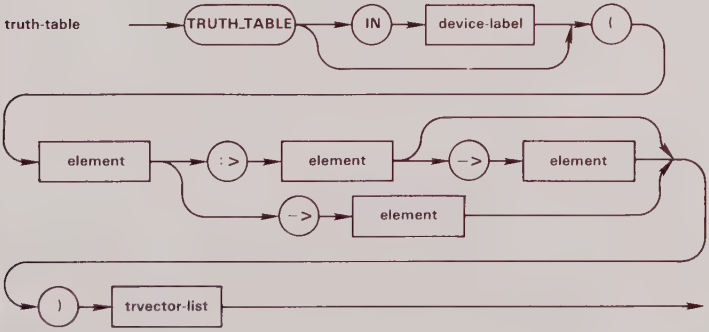

equation
!
ENABLE
lvalue
assign-op
expression
;

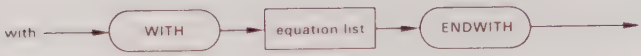

equation-
list
equation
;


equation
block
EQUATIONS
IN
device-label
equation-list

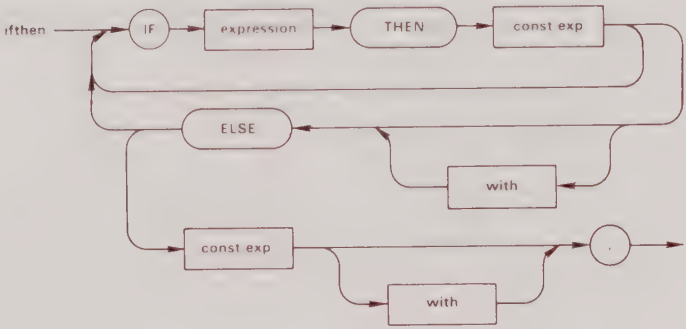

trvector
;
const-expr
: >
const-expr
->
const-expr
->
const-expr

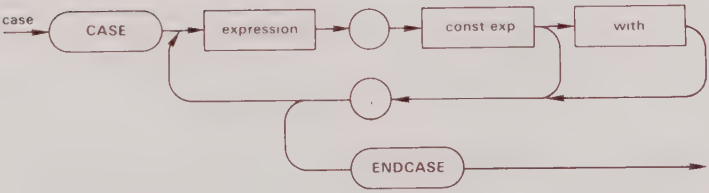

trvector-list
trvector
;

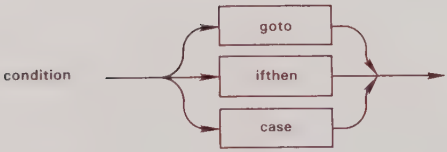

truth-table
TRUTH_TABLE
IN
device-label
(
element
: >
element
->
element
->
element
)
trvector-list

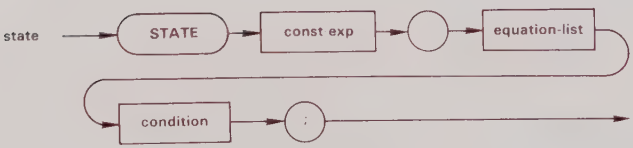

with
WITH
equation list
ENDWITH

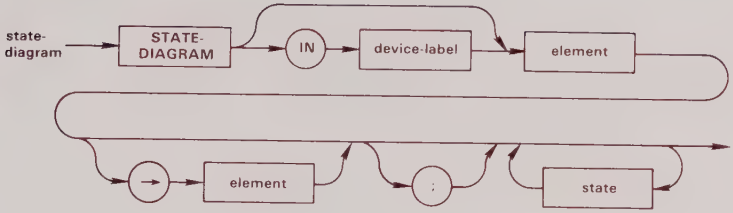
goto
GOTO
const exp
with

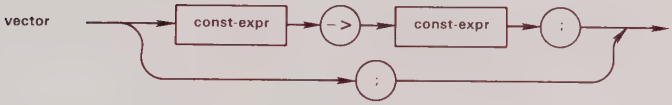

ifthen
IF
expression
THEN
const exp
ELSE
with
const exp
with


case
CASE
expression
const exp
with
ENDCASE

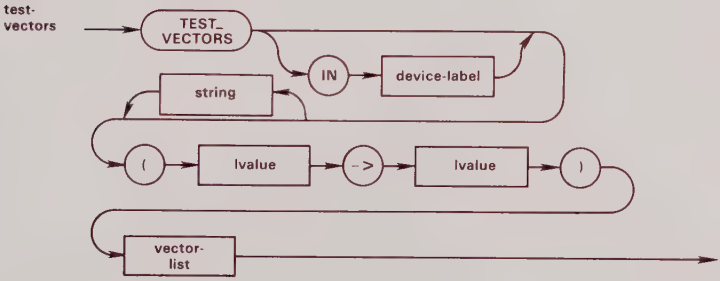

condition
goto
ifthen
case


state
STATE
const exp
equation-list
condition


state-
diagram
STATE-
DIAGRAM
IN
device-label
element
element
;
state


vector
const-expr
->
const-expr
;
;

vector-list
vector


test-
vectors
TEST_
VECTORS
IN
device-label
string
(
lvalue
->
lvalue
)
vector-
list

compiler directive
radix
include
repeat
irp
irpc
ifiden
ifidex
ifb
ifnb
ifdef
ifndef
if
message
alternate
standard
expr
const
page
exit

alternate

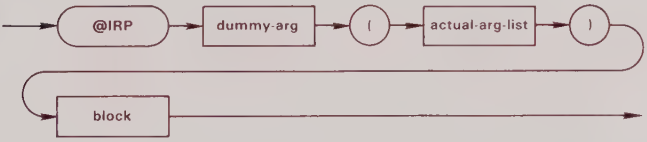
const

exit

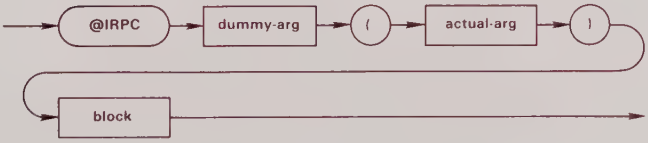
expr

include

irp

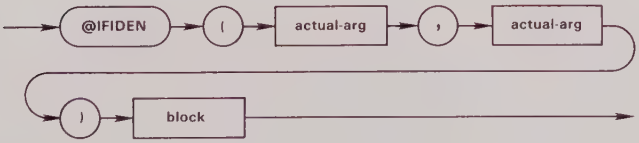


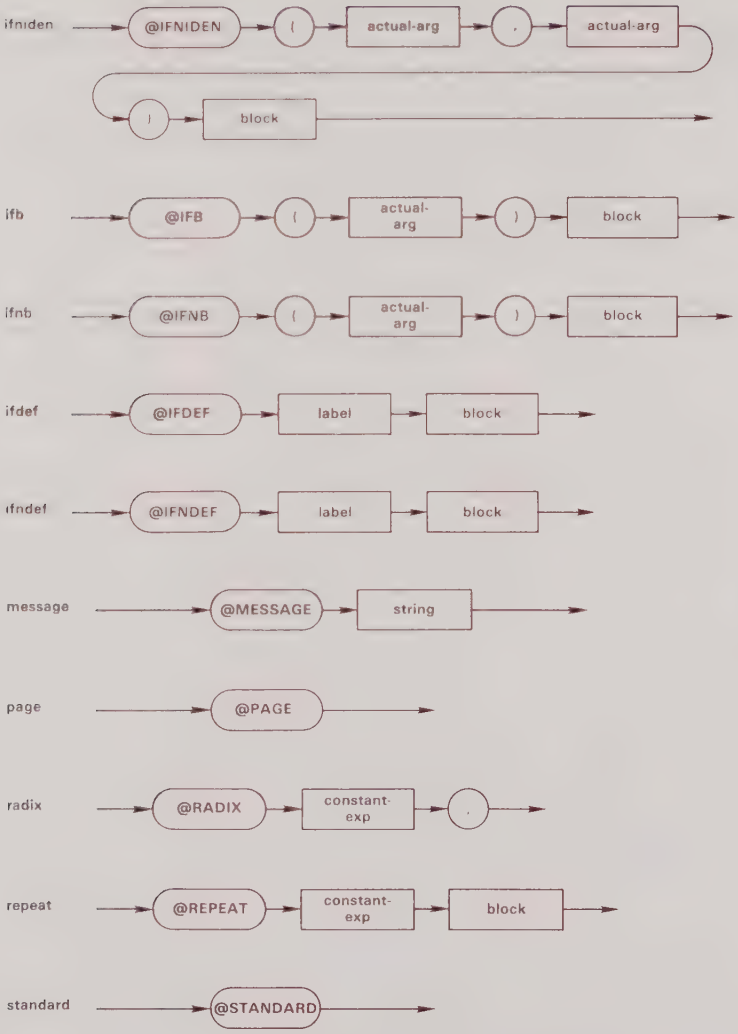
irpc



if

ifiden




ifniden
@IFNIDEN
(
actual-arg
,
actual-arg
)
block
ifb
@IFB
(
actual-arg
)
block
ifnb
@IFNB
(
actual-arg
)
block
ifdef
@IFDEF
label
block
ifndef
@IFNDEF
label
block
message
@MESSAGE
string
page
@PAGE
radix
@RADIX
constant-exp
.
repeat
@REPEAT
constant-exp
block
standard
@STANDARD

ASCII in Decimal and Hexadecimal

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	00	NUL	43	2B	+	86	56	V
1	01	SOH	44	2C	,	87	57	W
2	02	STX	45	2D	-	88	58	X
3	03	ETX	46	2E	.	89	59	Y
4	04	EOT	47	2F	/	90	5A	Z
5	05	ENQ	48	30	0	91	5B	[
6	06	ACK	49	31	1	92	5C	/
7	07	BEL	50	32	2	93	5D	]
8	08	BS	51	33	3	94	5E	^
9	09	HT	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	'
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	SP	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	|
39	27	'	82	52	R	125	7D	}
40	28	(	83	53	S	126	7E	~
41	29	)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

Set operations are applied according to normal rules of Boolean algebra, as described below. In the rules, upper-case letters are used to denote set names, and lower-case letters denote elements of a set. The letters, *k* and *n* are used as subscripts to the elements and to the sets. A subscript following a set name (upper-case letter) indicates how many elements the set contains. Thus, the notation, A_k, indicates that set *A* contains *k* elements. a_{k-1} is the *(k-1)*th element of set *A*. a_1 is the first element of set *A*.

$!A_k ::= [!a_k, !a_{k-1}, ..., !a_1]$

$-A_k ::= .!A_k + 1$

ENABLE $A_k ::=$ [ENABLE a_k, ENABLE a_{k-1},..., ENABLE a_1]

$A_k \& B_k ::= [a_k \& b_k, a_{k-1} \& b_{k-1}, ..., a_1 \& b_1]$

$A_k \# B_k ::= [a_k \# b_k, a_{k-1} \# b_{k-1}, ..., a1 \# b_1]$

$A_k \$ B_k ::= [a_k \$ b_k, a_{k-1} \$ b_{k-1}, ..., a_1 \$ b_1]$

$A_k !\$ B_k ::= [a_k !\$ b_k, a_{k-1} !\$ b_{k-1}, ..., a_1 !\$ b_1]$

$A_k == B_k ::= (a_k == b_k) \&$
$(a_{k-1} == b_{k-1}) \& ... \& (a_1 == b_1)$

$A_k != B_k ::= (a_k != b_k) \# (a_{k-1} != b_{k-1}) \# ... \# (a_1 != b_1)$

$A_k + B_k ::= D_k$
where:
$d_n ::= a_n \$ b_n \$ c_{n-1}$
$c_n ::= (a_n \& b_n) \# (a_n \& c_{n-1}) \# (b_n \& c_{n-1})$
$c0 ::= 0$

$A_k - B_k ::= A_k + (-B_k)$

$A_k < B_k ::= c_k$
where:
$c_n ::= (!a_n \& (b_n \# c_{n-1}) \# a_n \& b_n \& c_{n-1}) != 0$
$c0 ::= 0$

$A_k <= B_k ::= !(B_k < A_k)$

APPLICATIONS GUIDE

DATA I/O

TABLE OF CONTENTS

SECTION 1. INTRODUCTION

1.1 DESIGN EXAMPLE FILES 1-2
1.2 OTHER ABEL™ MANUALS 1-5
1.3 NOTATION CONVENTIONS 1-5

SECTION 2. DESIGN EXAMPLES

2.1 6809 MEMORY ADDRESS DECODER 2-2
2.2 12 TO 4 MULTIPLEXER 2-5
2.3 1 TO 8 DEMULTIPLEXER 2-8
2.4 4-BIT COUNTER/MULTIPLEXER 2-11
2.5 DECIMAL UP/DOWN COUNTER 2-17
2.6 8-BIT BARREL SHIFTER 2-22
2.7 7-SEGMENT DISPLAY DECODER 2-26
2.8 4-BIT COMPARATOR 2-30
2.9 BIDIRECTIONAL TRI-STATE BUFFER 2-36
2.9.1 Design Specification 2-36
2.9.2 Design Method 2-37

SECTION 3. DESIGN CONSIDERATIONS

3.1 USING STATE MACHINES 3-1
3.1.1 Use Identifiers Rather Than Numbers for States 3-2
3.1.2 "Power On" Register States 3-4
3.1.3 Design With the Registers, Not With the Outputs 3-5
3.1.4 Unsatisfied Transition Conditions Clear the State Register 3-5
3.1.5 Number Adjacent States for One-Bit Changes 3-6
3.1.6 Use State Register Outputs to Identify States 3-7
3.1.7 Combine the Use of Equations and State Diagrams 3-8
3.2 MULTIPLE ASSIGNMENTS TO THE SAME SIGNAL 3-9
3.3 HAZARD-FREE LOGIC DESIGN AND "REDUCE" 3-12
3.4 PASSING ARGUMENTS FROM THE COMMAND LINE 3-14
3.5 EFFECT OF POLARITY ON REDUCTION SPEED 3-18

SECTION 4. SIMULATION

4.1 TEST VECTORS AND SIMULATION 4-2
4.2 TRACE LEVELS AND BREAKPOINTS 4-2
4.3 DEBUGGING STATE MACHINES 4-4
4.4 MULTIPLE TEST VECTOR TABLES 4-5
4.5 USING MACROS AND DIRECTIVES TO CREATE TEST VECTORS 4-6
4.6 DON'T CARES IN SIMULATION 4-11
4.7 PRESET AND PRELOAD REGISTERS 4-13
4.7.1 Special Preset Considerations 4-14
4.7.2 TTL Preload 4-17
4.7.3 Supervoltage Preload 4-19
4.7.4 Preset/Reset Controlled by Product Term . . . 4-22
4.8 ASYNCHRONOUS CIRCUITS 4-24

SECTION 5. USING ADVANCED DEVICE FEATURES

5.1 OUTPUT ENABLES 5-1
5.1.1 Pin Controlled Output Enable 5-2
5.1.2 Term Controlled Output Enable 5-3
5.1.3 Configurable Output Enable 5-4
5.2 OUTPUT MACRO CELL CONTROL WITH ISTYPE 5-6
5.2.1 Controlling Macro Cell Polarity 5-6
5.2.2 Controlling Macro Cell Feedback Point . . . 5-8
5.2.3 Controlling Macro Cell Type 5-10
5.2.4 Controlling Register Type 5-10
5.3 NODES 5-11
5.3.1 Using Node Numbers 5-13
5.3.2 Using Shorthand Notation for Nodes 5-15
5.4 USING FLIP-FLOPS (MACRO CELL DEVICES) 5-17
5.4.1 The 82S159 Logic Diagram 5-18
5.4.2 Examination of the Source File 5-20
5.4.3 Programming the Flip-Flop Mode Control Fuses 5-22
5.4.4 Combining Equations 5-25
5.4.5 Specifying the Flip-Flop Inputs 5-26
5.4.6 Test Vectors 5-29
5.5 USING TRANSITION EQUATIONS 5-31

SECTION 6. ADVANCED DESIGNS

6.1 BLACKJACK MACHINE 6-2
6.2 6809 SYSTEM CONTROLLER 6-21

FIGURES

2-1 Block Diagram: 6809 Memory Address Decoder 2-2
2-2 Simplified Block Diagram: 6809 Memory Address Decoder 2-3
2-3 Block Diagram: 12 to 4 Multiplexer 2-5
2-4 Simplified Block Diagram: 12 to 4 Multiplexer 2-6
2-5 Block Diagram: 1 to 8 Demultiplexer 2-8
2-6 Simplified Block Diagram: 1 to 8 Demultiplexer 2-9
2-7 Block Diagram: 4-Bit Counter With 2-Input Multiplexer 2-11
2-8 Simplified Block Diagram: 4-Bit Counter With a 2-Input Multiplexer 2-12
2-9 Decimal Up/Down Counter With Synchronous Clear 2-18
2-10 Block Diagram: 8-Bit Barrel Shifter 2-22
2-11 Simplified Block Diagram: 8-Bit Barrel Shifter 2-23
2-12 Block Diagram: 7-Segment Display Decoder 2-27
2-13 Simplified Block Diagram: 7-Segment Display Decoder 2-27
2-14 Block Diagram: 4-Bit Comparator 2-30
2-15 Simplified Block Diagram: 4-Bit Comparator 2-31
3-1 Circuit Using an Input and its Complement 3-12
3-2 Timing Diagram for F = B & !C – !A & C 3-13
4-1 Timing Diagram Showing Test Vector Action 4-16
4-2 Internal Register of the 82S159 4-18
4-3 A Cross-Coupled Flip-Flop 4-25
5-1 Output Enable Controlled by Device Pin 5-2
5-2 Output Enable Controlled by Product Term 5-3
5-3 Configurable Output Enable 5-5
5-4 Controlling Macro Cells with ISTYPE 5-7
5-5 Logic Diagram with Node Numbers 5-12
5-6 Counter/Shifter Block Diagram 5-17
5-7 Logic Diagram of the 82S159 FPLS (Partial) 5-19
5-8 Flip-Flop Input Notation Examples 5-28
5-9 Abbreviated 82S105 Schematic 5-34
6-1 Blackjack Machine 6-2
6-2 Schematic, Blackjack Machine 6-4
6-3 Pictorial State Diagram: Blackjack Machine 6-7
6-4 System Schematic, 6809 Microprocessor System . . . 6-23
6-5 Timing Diagram: 6809 Dynamic RAM Access 6-25

LISTINGS

2-1 Source File 6809 Memory Address Decoder . . . 2-4
2-2 Source File: 12 to 4 Multiplexer 2-7
2-3 Source File: 1 to 8 Demultiplexer 2-10
2-4 Source File 4-Bit Counter With 2 Input Multiplexer . 2-15
2-5 Source File: Decimal Up/Down Counter With Synchronous Clear 2-20
2-6 Source File: 8-Bit Barrel Shifter 2-25
2-7 Source File: 7-Segment Display Decoder 2-29
2-8 Source File: 4-Bit Comparator 2-32
3-1 Source File: Decimal Counter 3-3
3-2 Multiple Equations Sections, 4-Bit Counter 3-10
3-3 Device Type Passed From Command Line, Memory Address Decoder 3-17
4-1 Source File With Multiple Test Vectors Sections 4-5
4-2 Test Vectors Described With a Macro and @IF and @IRP Directives: Memory Address Decoder . . . 4-6
4-3 PARSE Output Showing Expanded Output for a Macro and @IF and @IRP Directives: Memory Address Decoder 4-8
4-4 Test Vectors Described With a Macro, @CONST and @REPEAT Directives: Memory Address Decoder 4-9
4-5 PARSE Output Showing Expanded Output For a Macro and @CONST and @REPEAT Directives: Memory Address Decoder 4-10
4-6 Assignment of Don't Care Value (.x.) to Design Outputs 4-12
4-7 SIMULATE Results with Outputs Specified as Don't Care 4-12
4-8 Test Vectors for Special Preset Functions 4-15
4-9 Invoking the TTL Preload Function 4-18
4-10 The Illegal States Defined 4-20
4-11 Test Vectors for Illegal States 4-21
4-12 Using Test Vectors to Preload a State Machine 4-21
4-13 Controlling Reset/Preset by Product Term 4-23
4-14 Using the Input Side of the Test Vectors to Define Outputs 4-25
5-1 Controlling Macro Cells with ISTYPE Statements 5-9
5-2 Using Product Terms to Control Reset and Preset Nodes 5-14

5-3 Using Shorthand Notations to Control Nodes 5-16
5-4 A Shifter/Counter in an F82S159 5-23
5-5 SIMULATE Output (Partial) for the Shifter/Counter . . . 5-30
5-6 Transition Equations for a Decade Counter 5-33
6-1 State Declarations. This file is included where needed in modules with the @INCLUDE directive. 6-9
6-2 Source File: LED1, Low-Order Display Decoder 6-11
6-3 Source File: LED2, High-Order Display Decoder 6-13
6-4 Source File: ADD5, Five-Bit Ripple Adder 6-15
6-5 Source File: CARDSEL, Card Selector 6-17
6-6 Source File: CONTROL, State Machine Controller . . 6-19
6-7 Source File: 6809 Microprocessor Memory Controller 6-27

TABLES

1-1 Design Examples Supplied with ABEL™ 1-3
2-1 Address Ranges for the 6809 Controller 2-3
2-2 Counter Modes 2-12
6-1 Logic Designs Contained Within Advanced Examples 6-1
6-2 Programmable Devices and Their Function in the Blackjack Machine 6-3
6-3 States of the Blackjack State Machine 6-6
6-4 Address Ranges for a 6809 Microprocessor System 6-21
6-5 Design Approaches for Multiplexing and Refresh 6-22

1 INTRODUCTION

ABEL™ is a powerful tool for creating and testing programmable logic designs. Designs for a variety of logic families (PALs, PROMs, FPLAs, FPLSs, for example) can be described using the same general logic syntax. Logic reduction and simulation can be used to reduce the number of product terms required by a design, and to test a function. Designs described with ABEL are converted to programmer load files that are used with a logic programmer to program and test the real device.

DESIGN EXAMPLE FILES 1.1

This manual shows complete logic designs described with ABEL and discusses general topics that you will find useful as you use ABEL. All the logic designs and design features discussed in this manual are contained on the Design Examples disk or tape you received with the ABEL package; and are listed in table 1-1.

You can process these design examples with ABEL, either as they stand, or with your own modifications, to create programmer load files. Many of the design examples listed in table 1-1 are described in detail within this manual, and the section reference is given for these design examples. Design examples not described in the manual, but listed in table 1-1, can be examined as the need arises.

Table 1-1 lists the design examples alphabetically by filename of the ABEL source file and contains columns for:

- THE NAME OF THE SOURCE FILE. The source file uses the .ABL file extension (e.g., ADD5 is ADD5.ABL) although ABEL does not require that the file extension be entered when running ABEL on the source file.
- PROGRAMMABLE LOGIC DEVICE TYPE. The type of programmable logic device type used in the design example is listed to assist you in locating meaningful design examples.
- SECTION REFERENCE. Where applicable, a reference to a specific section in this manual is given that provides some detailed information.
- TYPE OF EXAMPLE. A brief description of the design example.

Table 1–1. Design Examples* Supplied with ABEL

Filename (*.ABL)	Device Type	Section Reference.	Type of Example
ADD5	F82S159	N/A	Macros for CARDSEL.ABL
ALARM	P16R4	N/A	Macros (state diagram)
ALTERA	EP300	5.2	Output macro cells
BARREL	P20R8	2.6	Sets
BCD7PAL	P16L8	N/A	PALASM example for TOABEL
BCD7ROM	RA5P8	2.7	Truth table
BGATES	P12H6	N/A	ABEL version of PALASM basic gates
BGEQN	F82S100	N/A	@repeat directive used w/equations
BJSTATE.TAB	N/A	N/A	Include file for CARDSEL.ABL
BUFFER	F82S153	2.9	Tri-state output control
CARDSEL	P16HD8	6.1	Blackjack machine
COMP4	P16HD8	2.8	Macros
COMP4ROM	RA8P4	N/A	Relational operators
CONTROL	P16R6	6.1	Blackjack state machine
COUNT10	P16R4	2.5	State Diagram
COUNT10A	P16R4	3.2	Equations and state diagram
COUNT4	P16R4	2.4	Equations
COUNT4A	P16R4	3.3	Multiple equations for one output
DECADE	F82S105	5.5	Transition equations, complement array
DMUX1T8	P16L8	2.3	Sets
FEEDBACK	P16HD8	N/A	Simulator example
FPLS	F82S157	N/A	Logic sequencer features
GATES	P12H6	N/A	PALASM example for TOABEL
LCUCROM	RA8P8	N/A	Macros, compiler directives

Table 1-1. Design Examples* Supplied with ABEL (Cont.)

Filename (*.ABL)	Device Type	Section Reference	Type of Example
LED1	F82S153	6.1	Truth tables for CARDSEL.ABL
LED2	F82S153	6.1	Truth tables for CARDSEL.ABL
M6809A	P14L4	2.1	Sets, relational operators
M6809B	P14L4	4.5	Macros with test vectors
M6809C	P14L4	4.5	Macros with test vectors
M6809D	P16HD8	3.5	Device type from command line
M6809ER	P14L4	N/A	Demonstrates simulation error
M6809ERR	P14L4	N/A	Demonstrates syntax error
MUX12T4	P14H4	2.2	Sets
OCTAL	P20X8	N/A	Exclusive OR equations
REF6809	P16L8	6.2	Multiple equation and vectors
REGFB	P16R4	N/A	Simulator example
SHIFT40	32R16	N/A	84-pin PAL example
TI1SHOT	P19R8	N/A	Input registers
TRAFFIC	F82S167	N/A	State machine RS flip/flops
SHIFTCNT	F82S159	5.2	Controlling JK and D flip/flops
VTI	E10P8	N/A	Programmable XOR

*This table represents a partial list of design examples available at the time of printing. To verify the design examples provided on your ABEL disk or tape, use the operating system to print the file named EXAMPLES.TXT.

OTHER ABEL MANUALS 1.2

This Applications Guide is one of three ABEL manuals; the other two manuals are:

ABEL Language Reference

This manuals contains a complete definition of the ABEL design language and is intended as a quick reference and a description of the ABEL design language.

ABEL User's Guide

This manual describes how the use the ABEL language processor. It also describes setup procedures and the basics of ABEL useage.

NOTATION CONVENTIONS 1.3

Italics are used to indicate references (by name) to items contained in the design examples, the illustrations, the listings, and the tables.

DESIGN EXAMPLES 2

This section contains a collection of logic design examples believed to be representative of typical programmable logic applications, or examples that serve to illustrate significant ABEL™ features. Use these examples both to learn more about ABEL™ and to start designing with ABEL™. We encourage you to use these examples directly by modifying them to suit your needs, or by incorporating them into larger system designs.

The titles of the designs indicate the logic design problem being solved, the major features of ABEL™ used to solve the problem, and the type of programmable logic device used. Each design comes complete with block diagrams and source file listings.

All the examples presented in this section are included on your ABEL™ distribution disk or tape. Additional examples not explained here are also included. A list and brief description of all examples distributed with ABEL™ can be found in the file named EXAMPLES.TXT on your distribution disk or tape; and also in table 1-1 in the preceding section (Use the operating system to display the list of design examples contained in EXAMPLES.TXT. EXAMPLES.TXT gives the name of the source file, the type of programmable logic device used, the name of the load file, and a brief description of the design example.)

6809 MEMORY ADDRESS DECODER (EQUATIONS/P14L4) 2.1

Address decoding is a typical application of programmable logic devices, and the following describes the ABEL™ implementation of such a design. The address decoder is part of the larger 6809 controller design described in section 6–2. This is also the same design found in the ABEL™ User's Guide.

Design Specification

Figure 2–1 shows the block diagram for this design and a continuous block of memory divided into sections containing dynamic RAM (*DRAM*), I/O (*IO*), and two sections of ROM (*ROM1* and *ROM2*). The purpose of this decoder is to monitor the 6 high-order bits (*A15–A10*) of a sixteen-bit address bus and select the correct section of memory based on the value of these address bits. To perform this function, a simple decoder with six inputs and four outputs is designed with a P14L4 PAL.

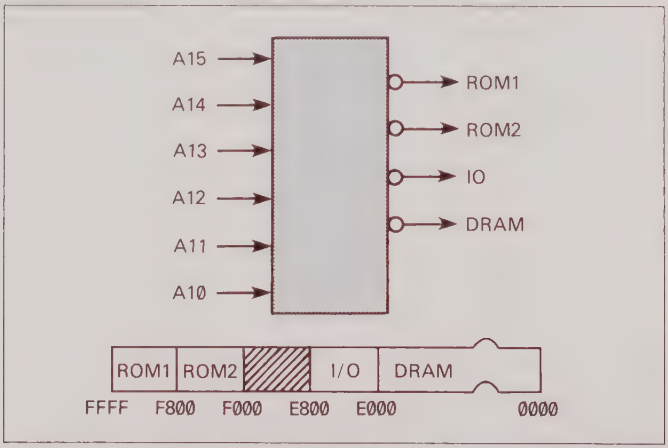


Figure 2–1. Block Diagram: 6809 Memory Address Decoder

Table 2–1 shows the address ranges associated with each section of memory. These address ranges can also be seen on figure 2–1, above.

Table 2-1. Address Ranges for the 6809 Controller

Memory Section	Address Range (hex)
DRAM	0000-DFFF
I/O	E000-E7FF
ROM2	F000-F7FF
ROM1	F800-FFFF

Design Method

Figure 2-2 shows a simplified block diagram for the address decoder. The address decoder is implemented with simple Boolean equations employing both relational and logical operators as shown in listing 2-1. A significant amount of simplification is achieved by grouping the address bits into a set named *Address*. The lower-order ten address bits that are not used for the address decode are given "don't care" values in the address set. In this way, the designer indicates that the address in the overall design (that beyond the decoder) contains sixteen bits, but that bits 0-9 do not affect the decode of that address. This is opposed to simply defining the set as, *Address* = [*A15,A14,A13,A12,A11,A10*], which ignores the existence of the lower-order bits. Specifying all 16 address lines as members of the address set also allows full 16-bit comparisons of the address value against the ranges shown in table 2-1.

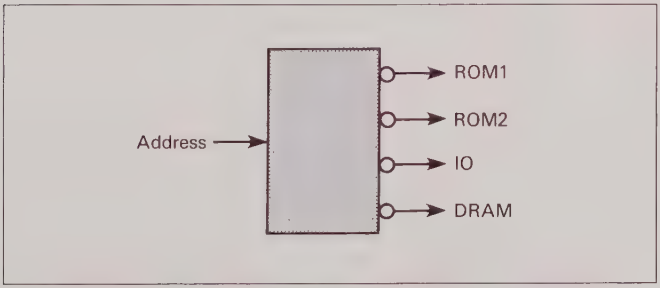


Figure 2-2. Simplified Block Diagram: 6809 Memory Address Decoder

Test Vectors

In this design, the test vectors are a straightforward listing of the values that must appear on the output lines for specific address values. The address values are specified in hexadecimal notation.

```
module m6809a
title '6809 memory decode
Jean Designer     Data I/O Corp Redmond WA   24 Feb 1984'

                        U09      device   'P14L4';
        A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
        ROM1,IO,ROM2,DRAM       pin 14,15,16,17;

        H,L,X    = 1,0,.X.;
        Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

equations
        !DRAM    = (Address <= ^hDFFF);

        !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);

        !ROM2    = (Address >= ^hF000) & (Address <= ^hF7FF);

        !ROM1    = (Address >= ^hF800);

test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
               ^h0000  -> [  H,   H,   H,   L ];
               ^h4000  -> [  H,   H,   H,   L ];
               ^h8000  -> [  H,   H,   H,   L ];
               ^hC000  -> [  H,   H,   H,   L ];
               ^hE000  -> [  H,   H,   L,   H ];
               ^hE800  -> [  H,   H,   H,   H ];
               ^hF000  -> [  H,   L,   H,   H ];
               ^hF800  -> [  L,   H,   H,   H ];
end m6809a
```

Listing 2-1. Source File: 6809 Memory Address Decoder

12 TO 4 MULTIPLEXER (EQUATIONS/P14H4) 2.2

Following is a description of a 12-input to 4-output multiplexer implemented with one Boolean equation.

Design Specification

Figure 2-3 shows the block diagram for this design. The multiplexer selects one of three sets of four inputs and routes that set to the outputs. The inputs are *a0 - a3*, *b0 - b3*, and c0 - c3. The outputs are *y0-y3*. The routing of inputs to outputs is straightforward: *a0* or *b0* or *co* is routed to the output *y0*, *a1* or *b1* or *c1* is routed to the output *y1*, and so on with the remaining outputs.

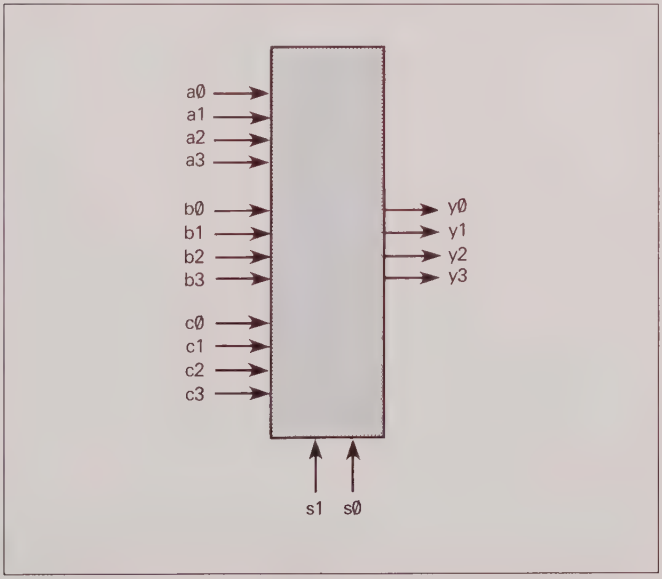


Figure 2-3. Block Diagram: 12 to 4 Multiplexer

Design Method

Figure 2–4 shows a block diagram for the same multiplexer after sets have been used to group the signals. All of the inputs have been grouped into the sets *a*, *b*, and *c*; the outputs and select lines are grouped into the sets, *y* and *select*, respectively. This grouping of signals into sets takes place in the declaration section of the source file listed in listing 2–2.

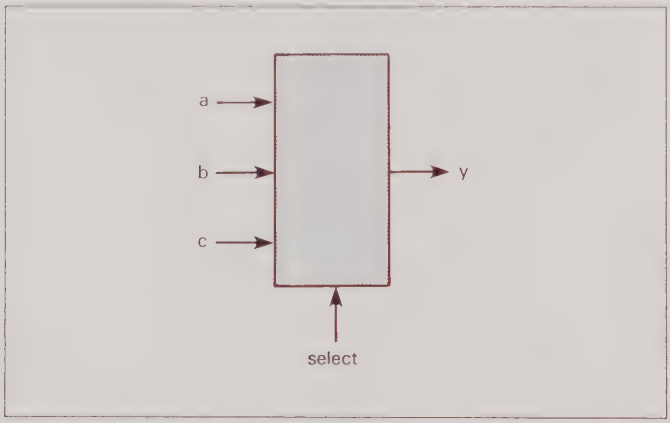


Figure 2–4. Simplified Block Diagram: 12 to 4 Multiplexer

Once the sets have been declared, specification of the design is easy, requiring only one equation. The equation is expressed in sum of products form, with each product being the logical AND of the select lines and the associated inputs. Take for example the first product:

```
(select == 0) & a
```

The relational expression inside the parentheses produces an expression that evaluates to true or false value depending on the values of *s1* and *s0*. This expression is then ANDed with the set *a* which contains the four bits, *a0* – *a3*. Assume that *select* does equal 0 (*s1* = 0 and *s0* = 0) so that a true value is produced. The true is then ANDed with the set *a* on a bit by bit basis, which in effect sets the product term to *a*. If *select* were not equal to 0, the relational expression inside the parentheses would produce a false value, which when ANDed with anything, would give all zeroes. The other product terms in the

equation work in the same manner. Because *select* takes on only one value at a time, only one of the product terms passes the value of an input set along to the output set; the others contribute 0 bits to the ORs.

Test Vectors

The test vectors for this design are specified in terms of the input, output and select sets. Note that the values for a set can be specified by decimal numbers and by other sets. The constants *H* and *L* used in the test vectors were declared as four bit sets containing all ones or all zeroes.

```
module mux12t4
title '12 to 4 multiplexer   24 Feb 1984
Charles Olivier & Dave Pellerin  Data I/O Corp.  Redmond WA'

"declarations

        IC1             device  'P14H4';

        a0,a1,a2,a3     pin     1,2,3,4;
        b0,b1,b2,b3     pin     5,6,7,8;
        c0,c1,c2,c3     pin     9,11,12,13;

"define select inputs

        s1,s0   pin     18,19;

"define output

        y0,y1,y2,y3     pin     14,15,16,17;

        H       =       [1,1,1,1];
        L       =       [0,0,0,0];
        X       =       [.x.,.x.,.x.,.x.];
        select  =       [s1, s0];
        y       =       [y3,y2,y1,y0];
        a       =       [a3,a2,a1,a0];
        b       =       [b3,b2,b1,b0];
        c       =       [c3,c2,c1,c0];

equations

        y  = (select == 0) & a #
             (select == 1) & b #
             (select == 2) & c #
             (select == 3) & c ;

test_vectors ([select, a, b, c] -> y)
              [0      , 1, X, X] -> 1; "select = 0,gates line a to output
              [0      ,10, H, L] -> 10;
              [0      , 5, H, L] -> 5;

              [1      , H, 3, H] -> 3; "select = 1,gates line b to output
              [1      ,10, 7, H] -> 7;
              [1      , L,15, L] -> 15;

              [2      , L, L, 8] -> 8; "select = 2,gates lines c to output
              [2      , H, H, 9] -> 9;
              [2      , L, L, 1] -> 1;

              [3      , H, H, 0] -> 0; "select = 3,gates lines c to output
              [3      , L, L, 9] -> 9;
              [3      , H, L, 0] -> 0;

end mux12t4
```

Listing 2-2. Source File: 12 to 4 Multiplexer

1 TO 8 DEMULTIPLEXER (EQUATIONS/P16L8) 2.3

Following is the design for a simple one line to eight line demultiplexer with an enable. The design is implemented with Boolean equations and a P16L8 PAL.

Design Specification

Figure 2–5 shows a block diagram for a one line to eight line demultiplexer with active low outputs, a one bit input, three select lines and an enable. The demultiplexer routes the input named *data* to one of the eight output lines, *y0–y7*, according to the value present on the select lines, *s0–s2*. Because the outputs are inverted, they show the inverse of the input line. The select lines carry a three bit binary number whose value ranges from 0 to 7, thus selecting one of the outputs. *enb* is an enable line. When *enb* is low (0), the outputs go to high impedance. When *enb* is high, the outputs are determined by the demultiplexing function.

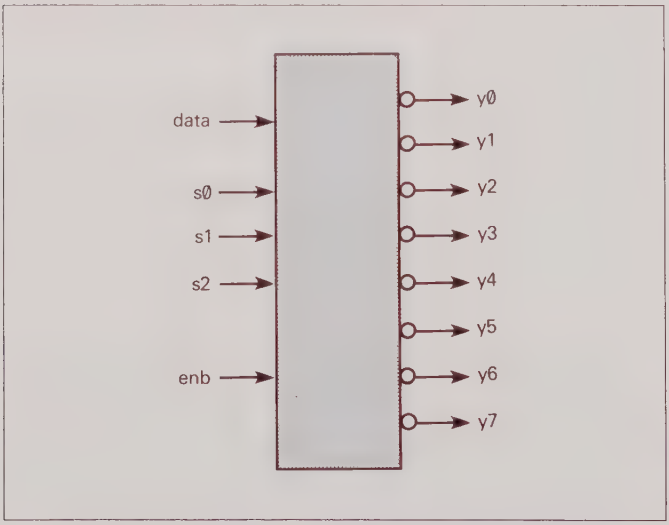


Figure 2–5. Block Diagram: 1 to 8 Demultiplexer

Design Method

Figure 2-6 shows a simplified block diagram of the demultiplexer. The select lines and outputs are collected into sets named *select* and *outputs* to simplify the Boolean equations with which the demultiplexing function is described. The ABEL™ description of this logic is shown in listing 2–3. Each output is determined by ANDing the data line with a relational expression that checks for equivalence between the select lines and one of the eight possible select values. Because *select* can have only one value at any given time, only one of the outputs is selected for an AND with the data value. The selected output shows the inverse of the data line.

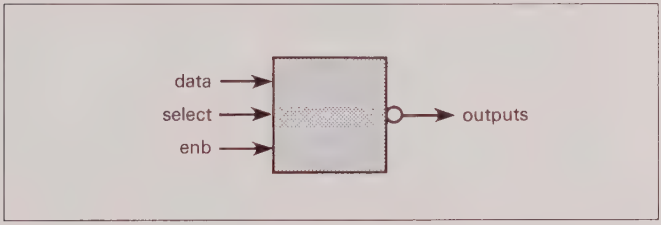


Figure 2–6. Simplified Block Diagram: 1 to 8 Demultiplexer

The enable function is implemented with a separate equation using the set *outputs* along with the ENABLE keyword. Thus the equation,

```
enable outputs = enb ;
```

assigns the enables on each of the output lines to the input, *enb*.

Test Vectors

The test vectors for this design first select each of the outputs with the data input and enable high, then they select each of the outputs with the data input low and the enable high. Finally, the enable is checked by setting it low; all the outputs go to high impedance.

```
module dmux1t8
title '1 to 8 line demultiplexer
Charles Olivier   Data I/O Corp. Redmond WA   24 Feb 1984'

"Description
" This is an example of a 1 to 8 line demultiplexer using a PAL16L8.
" When the enable line goes low, the outputs will be high impedance.

"declarations
                IC1     device  'P16L8';

        y0,y1,y2,y3     pin     12,13,14,15;
        y4,y5,y6,y7     pin     16,17,18,19;
        s0,s1,s2        pin     1,2,3;
        data,enb        pin     4,5;

        H,L,Z   =       1,0,.Z.;
        select  =       [s2, s1, s0];
        outputs =       [y7, y6, y5, y4, y3, y2, y1, y0];

equations
        !y0     = (select == 0) & data;
        !y1     = (select == 1) & data;
        !y2     = (select == 2) & data;
        !y3     = (select == 3) & data;
        !y4     = (select == 4) & data;
        !y5     = (select == 5) & data;
        !y6     = (select == 6) & data;
        !y7     = (select == 7) & data;

 enable outputs = enb;

test_vectors  'Test the demultiplexer with a high input'
        ([enb,select,data] -> [y7,y6,y5,y4,y3,y2,y1,y0])
             [H, 0, H]      -> [H, H, H, H, H, H, H, L];  "Select y0
             [H, 1, H]      -> [H, H, H, H, H, H, L, H];  "Select y1
             [H, 2, H]      -> [H, H, H, H, H, L, H, H];  "Select y2
             [H, 3, H]      -> [H, H, H, H, L, H, H, H];  "Select y3
             [H, 4, H]      -> [H, H, H, L, H, H, H, H];  "Select y4
             [H, 5, H]      -> [H, H, L, H, H, H, H, H];  "Select y5
             [H, 6, H]      -> [H, L, H, H, H, H, H, H];  "Select y6
             [H, 7, H]      -> [L, H, H, H, H, H, H, H];  "Select y7

test_vectors  'Test the demultiplexer with a low input'
        ([enb,select,data] -> [y7,y6,y5,y4,y3,y2,y1,y0])
             [H, 0, L]      -> [H, H, H, H, H, H, H, H];  "Select y0
             [H, 1, L]      -> [H, H, H, H, H, H, H, H];  "Select y1
             [H, 2, L]      -> [H, H, H, H, H, H, H, H];  "Select y2
             [H, 3, L]      -> [H, H, H, H, H, H, H, H];  "Select y3
             [H, 4, L]      -> [H, H, H, H, H, H, H, H];  "Select y4
             [H, 5, L]      -> [H, H, H, H, H, H, H, H];  "Select y5
             [H, 6, L]      -> [H, H, H, H, H, H, H, H];  "Select y6
             [H, 7, L]      -> [H, H, H, H, H, H, H, H];  "Select y7

test_vectors  'Test the enable input'
        ([enb,select,data] -> [y7,y6,y5,y4,y3,y2,y1,y0])
             [L, 0, H]      -> [Z, Z, Z, Z, Z, Z, Z, Z];
             [L, 0, L]      -> [Z, Z, Z, Z, Z, Z, Z, Z];
end dmux1t8
```

Listing 2-3. 1 to 8 Demultiplexer

4-BIT COUNTER/MULTIPLEXER (EQUATIONS/P16R4) 2.4

Following is a design for a 4-bit synchronous counter implemented with a P16R4. Counter features include carry-in and carry-out, 2 input multiplexing and a hold state. The counter is described by Boolean equations.

Design Specification

Figure 2-7 shows the counter and its signals. The outputs, *Q0*, *Q1*, *Q2*, and *Q3* show the current count, with *Q0* being the low-order bit and *Q3* being the high-order bit. The counter has four different modes of operation: hold, load A, load B, and increment. The modes are selected by the inputs, *I0* and *I1*, as indicated in table 2-2. In the hold mode, the current count is retained regardless of clocking. When in load A mode, the counter loads the values on *A0-A3* on the next clock pulse. Similarly, the *B0-B3* inputs are loaded into the counter on the next clock pulse when the mode is load B. In increment mode, the count is increased by the value on the carry-in line *CI* on a clock pulse. If the count overflows from 15 (hex F) to 0, the carry-out line *CO*, goes to 1. The output control, *OC*, enables the outputs when low (*OC* =0) and forces the outputs to high impedance when high (*OC* =1).

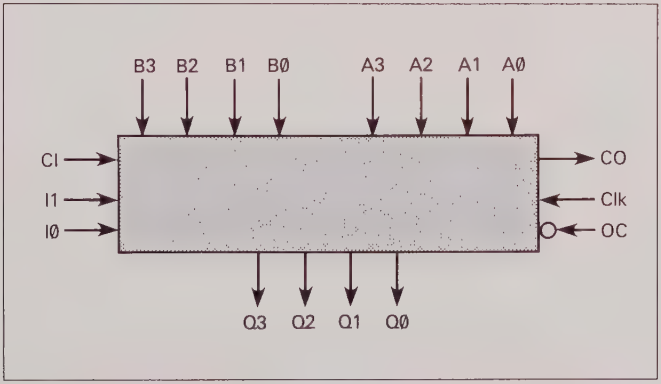


Figure 2-7. Block Diagram: 4-Bit Counter With 2 Input Multiplexer

Table 2-2. Counter Modes

Mode	I1	I0	Description
Hold	0	0	count remains unchanged
Load A	0	1	load A0–A3 into the count registers
Load B	1	0	load B0–B3 into the count registers
Increment	1	1	increment the count by 1 on clock pulse

The carry-in and carry-out lines operate such that two or more of the counters can be chained together to form a wider counter. To do this, the carry-out of one counter is connected to the carry in of the next counter. Thus, when the first counter counts to 16, it is cleared to 0 and its carry-out bit is one, causing the next counter's increment to be one.

Design Method

The counter is described with Boolean equations. Figure 2-8 shows the simplified block diagram corresponding to the ABEL™ implementation of the counter design. Listing 2-4 shows the source file.

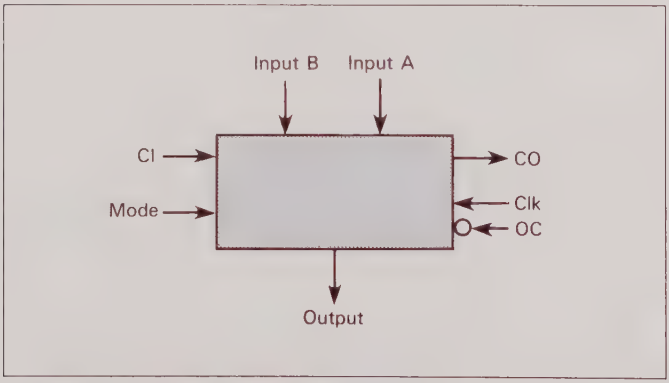


Figure 2-8. Simplified Block Diagram: 4-Bit Counter With 2 Input Multiplexer

The design is simplified by grouping the input bits, output bits and mode selectors into sets, so that they can be referenced by name. The inputs *A0–A3* are assigned to the set *InputA*, the inputs *B0–B3* to the set *InputB*, the outputs *Q0–Q3* and *CO* to the set *Output*, and the mode selectors *I0* and *I1* to the set *Mode*.

Notation in the source file can be further simplified by some simple constant assignments. *H* represents a 1, *L* represents a 0, *X* represents the special constant .X., and so on. The four possible modes are also assigned as constants: *Hold* = 0, *LoadA* = 1, *LoadB* = 2, and *Incr* = 3. These assignments correspond to the decimal equivalent of the two bit binary number formed of *I1* and *I0*, as already shown in table 2–2. Note also that this two bit binary number is the set *Mode*.

Thus, in the equations section of the source file, the set *Mode* is compared to the different possible modes--by name. Take for example, the expression:

```
Mode ==LoadA
```

This is equivalent to:

```
[I1,I0] == 1
```

And this is equivalent to:

```
(I1 ==0) & (I0 == 1)
```

which can be written as:

```
! I1 & I0
```

This means "if I1 is low and I0 is high." By using sets and constant assignments, the source file becomes more meaningful. The expression, *Mode ==LoadA*, can be read as, "if the mode is LoadA."

The equations section of the source file simply describes the Boolean equations for a 4-bit adder with carry-in, carry-out and multiplexed load.

Test Vectors

The advantages of set notation become even more apparent in the test vectors section of the source file. In the test vectors section, it is necessary to show the required output for various given inputs. Rather than listing the outputs and inputs bit by bit, advantage is taken of sets, constants, and hexadecimal notation to simplify the vectors.

For example, because *LoadA* was assigned the constant value, 1, the name, *LoadA*, can be used directly in the test vectors as a value for *Mode*. Thus, it is unnecessary to remember that *Mode* is made up of *I1* and *I2* and that *LoadA* corresponds to *I1* = 0, *I0* = 1.

The test vectors shown in listing 2–4 test for proper loading of *InputA* and *InputB*, for increments after loads, for hold states, for correct operation of the carry–out, and for normal increment mode. Refer to the comments beside the test vectors for examples of each type of test.

The PAL 16R4 and many other devices have a dedicated output enable pin. This pin must be held at the proper level (0 or 1) to observe the outputs. The test vectors in listing 2–4 include the output enable pin (OC).

```
module count4
title '4-bit counter with 2 input mux
based on an example by Birkner/Coli in the MMI PAL Handbook
Dan Burrier     Data I/O  Redmond WA      24 Feb 1984'

        P7022    device 'P16R4';

        Clk,OC                          pin  1,11;
        A0,A1,A2,A3,B0,B1,B2,B3         pin  2,3,4,5,6,7,8,9;
        Q3,Q2,Q1,Q0                     pin  14,15,16,17;
        CO,I1,I0,CI                     pin  12,13,18,19;

                        H,L,X,Z,C       = 1,0, .X.,.Z.,.C.;
                        InputA          = [A3,A2,A1,A0];
                        InputB          = [B3,B2,B1,B0];
                        Output          = [CO,Q3,Q2,Q1,Q0];
                        Mode            = [I1,I0];

        Hold,LoadA,LoadB,Incr           = 0,1,2,3;  " define Modes

equations
        !Q0     := (Mode==Hold ) & !Q0
                 # (Mode==LoadA) & !A0
                 # (Mode==LoadB) & !B0
                 # (Mode==Incr ) & !CI & !Q0        "Hold if no carry
                 # (Mode==Incr ) &  CI &  Q0 ;

        !Q1     := (Mode==Hold ) & !Q1
                 # (Mode==LoadA) & !A1
                 # (Mode==LoadB) & !B1
                 # (Mode==Incr ) & !CI & !Q1        "Hold if no carry
                 # (Mode==Incr ) & !Q0 & !Q1        "Hold if Q0=L
                 # (Mode==Incr ) &  CI &  Q0 &  Q1 ;

        !Q2     := (Mode==Hold ) & !Q2
                 # (Mode==LoadA) & !A2
                 # (Mode==LoadB) & !B2
                 # (Mode==Incr ) & !CI & !Q2        "Hold if no carry
                 # (Mode==Incr ) & !Q0 & !Q2        "Hold if Q0=L
                 # (Mode==Incr ) & !Q1 & !Q2        "Hold if Q1=L
                 # (Mode==Incr ) &  CI &  Q0 &  Q1 &  Q2 ;

        !Q3     := (Mode==Hold ) & !Q3
                 # (Mode==LoadA) & !A3
                 # (Mode==LoadB) & !B3
                 # (Mode==Incr ) & !CI & !Q3        "Hold if no carry
                 # (Mode==Incr ) & !Q0 & !Q3        "Hold if Q0=L
                 # (Mode==Incr ) & !Q1 & !Q3        "Hold if Q1=L
                 # (Mode==Incr ) & !Q2 & !Q3        "Hold if Q2=L
                 # (Mode==Incr ) &  CI &  Q0 &  Q1 &  Q2 &  Q3 ;

        !CO     = !CI # !Q0 # !Q1 # !Q2 # !Q3 ;
```

Listing 2-4. Source file: 4-bit Counter with 2 Input Multiplexer

Listing continued on next page.

```
@page
test_vectors    ' test Load A and B'
        ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
          [ C, L, LoadA,  ^h0  ,  ^hF  ,X  ] ->  ^h0;
          [ C, L, LoadB,  ^h0  ,  ^hF  ,X  ] ->  ^hF;
            C, L, LoadA,  ^h1  ,  ^hF  ,X  ] ->  ^h1;
          [ C, L, LoadB,  ^h1  ,  ^h7  ,X  ] ->  ^h7;
          [ C, L, LoadA,  ^h2  ,  ^h6  ,X  ] ->  ^h2;
          [ C, L, LoadB,  ^h2  ,  ^h6  ,X  ] ->  ^h8;
          [ C, L, LoadA,  ^h4  ,  ^h4  ,X  ] ->  ^h4;
          [ C, L, LoadB,  ^h4  ,  ^h4  ,X  ] ->  ^hD;
          [ C, L, LoadA,  ^h8  ,  ^h8  ,X  ] ->  ^h8;
          [ C, L, LoadB,  ^h8  ,  ^h8  ,X  ] ->  ^hE;
          [ C, L, LoadA,  ^h0  ,  ^hF  ,X  ] ->  ^h0;
          [ C, L, LoadB,  ^h0  ,  ^hF  ,X  ] ->  ^hF;

test_vectors    ' test increment'
        ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
          [ C, L, LoadB,    X   ,  ^h1  ,X  ] ->  ^h1;
          [ C, L, Incr ,    X   ,    X  ,H  ] ->  ^h2;
          [ C, L, LoadB,    X   ,  ^h3  ,X  ] ->  ^h3;
            C, L, Incr ,    X   ,    X  ,H  ] ->  ^h4;
          [ C, L, LoadA,  ^h7   ,    X  ,X  ] ->  ^h7;
          [ C, L, Incr ,    X   ,    X  ,H  ] ->  ^h8;
          [ C, L, LoadA,  ^hF   ,    X  ,X  ] ->  ^hF;
            C, L, Incr ,    X   ,    X  ,H  ] ->  ^h0;       "roll over
            C, L, LoadB,    X   ,  ^hC  ,X  ] ->  ^hC;
          [ C, L, Incr ,    X   ,    X  ,H  ] ->  ^hD;
          [ C, L, Hold ,    X   ,    X  ,H  ] ->  ^hD;

test_vectors    ' test carry'
        ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
          [ C, L, Incr ,    X   ,    X  ,H  ] ->  ^hE;
          [ C, L, Incr ,    X   ,    X  ,H  ] ->  ^h1F;      "carry out
          [ C, L, Incr ,    X   ,    X  ,H  ] ->  ^h0;       "roll over
          [ C, L, Incr ,    X   ,    X  ,H  ] ->  ^h1;
          [ C, L, Incr ,    X   ,    X  ,L  ] ->  ^h1;       "no carry in
          [ C, L, Incr ,    X   ,    X  ,H  ] ->  ^h2;
          [ L, H, Hold ,    X   ,    X  ,X  ] ->  [X,Z,Z,Z,Z];
end count4
```

Listing 2–4. *(Continued)*

DECIMAL UP/DOWN COUNTER (STATE DIAGRAM/16R4) 2.5

The following design implements a bidirectional, up/down, decimal counter in a P16R4. The design is described with an ABEL™ state diagram.

Design Specification

Figure 2–9 shows the counter design and its accompanying state diagram. The active low (inverted) outputs Q0 – Q3 contain the current count, with Q0 being the low–order bit and Q3 being the high–order bit. The direction of the count (up or down) is determined by *Dir*. When *Dir* is high, the counter counts up by one on each clock pulse. When *Dir* is low, the counter counts down by one on each clock pulse. Valid counts are in the range 0–9. During an up count, the counter counts from 0 to 9 and back to zero. During a down count, the counter counts backwards from 9 to 0 and then to 9 again.

The outputs are enabled when *OE* is low; when *OE* is high, the outputs are set to high impedance. Note, however, that even if the outputs are disabled, the count is still increased or decreased on the clock pulse. This occurs because the true outputs of the PAL are buffered after the registers and the output enable controls these buffered outputs, not the registers. Thus, the registers change values independently of the true outputs.

Design Method

The counter is described as a state machine by a STATE_DIAGRAM section. Compare this to the Boolean equations that describe the four–bit counter in section 2.4. Listing 2–5 shows the ABEL™ source file for the counter.

In the source file, the design is given a title, the device type is specified, and pin declarations are made. The FLAG statement is used to cause level 3 reduction to be performed by the language processor. (This design will not fit the chosen P16R4 device if level 1 or level 0 reduction is performed.)

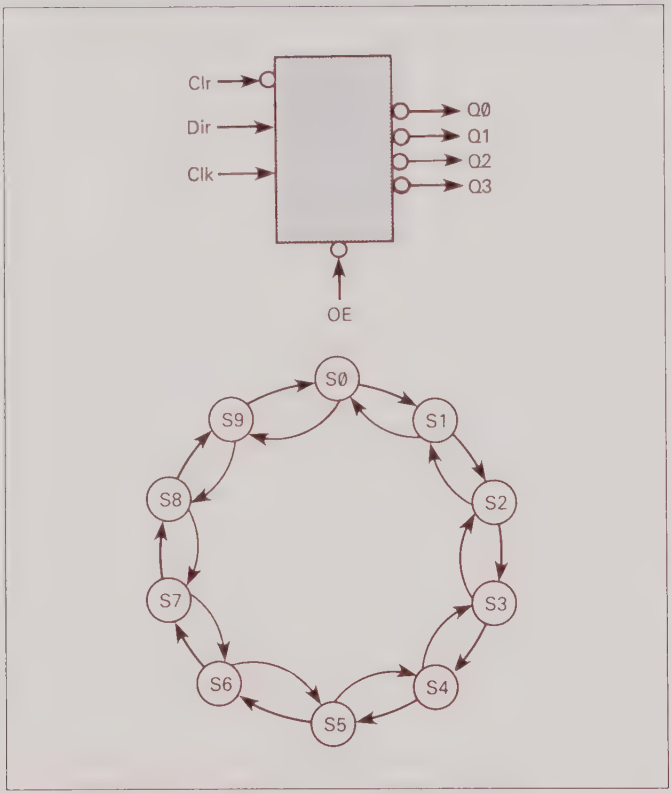


Figure 2-9. Decimal Up/Down Counter With Synchronous Clear

Constants are declared to simplify notation in the source. One constant for each possible state of the state machine is declared so that the states can be referenced by name. (This also allows changes to the state sequence to be made easily.) There are ten states corresponding to the ten possible counts. Because the outputs of our counter are active low, the constant *S0* is declared as the binary number ^b1111 (equivalent to 0000 in active-high

logic). Other states of the counter are declared in the same manner, so that each state corresponds to a decimal count. When the state machine is in state 3, the count is 3 (in active low logic); when it is in state 4, the count is four; and so on.

The inputs *Clr* and *Dir* are grouped into a set called *Mode* that determines the mode of the counter. The counter can be in *Up*, *Down*, or *Clear* modes. It is important to note that the fourth bit combination is not needed for a counter mode, but is assigned to clear because of the "don't care" (X) specified in the constant declaration for *Clear*. Using a set for the modes of the counter further simplifies the STATE_DIAGRAM.

The STATE_DIAGRAM begins with the STATE_DIAGRAM statement that names the state machine and the state register. The set, [*Q3, Q2, Q1,* Q0], is used to specify the state register, which also defines the design outputs.

Within the STATE_DIAGRAM, CASE statements are used to determine the state transitions. Simple comparisons between *Mode* and the three possible modes of operation are made to determine what the next step must be. Take for example, the part of the STATE_DIAGRAM that specifies the state transitions for *State 5*, repeated here:

```
State S5:     case  (Mode == Up)      : S6 ;
                    (Mode == Down)    : S4 ;
                    (Mode == Clear)   : S0 ;
              endcase;
```

This means that if we are in state S5 and the mode is *Up*, the next state is state S6; if the mode is *Down*, the next state is S4; and if the mode is *Clear*, the next state is S0. All the state transitions are described in a similar manner.

Test Vectors

The specification of the test vectors for this design is simple. The declared constants are used in the vectors along with the sets *Mode* and [*Q3,Q2,Q1,Q0*] to explicitly state the expected operation of the design. First, the clear function is tested, then the counter is cycled through an up count from 0 to 9, back to 0, and to 1, at which point the direction is reversed and the counter counts down from 1 to 0 and to 9. Next, the output enable is tested. When *OE* goes high (1), the outputs should be disabled and all be at high impedance, even though the count was decreased by one. Then on the next clock, with *OE* set back low, the count should be 7. A single Z in the high-impedance test vector expands to cover all outputs. That is, Z expands to [Z,Z,Z,Z].

```
module count10   flag '-r3'
title 'decimal Up/Down counter with synchronous Clear
Michael Holley      Data I/O Corp   24 Feb 1984'

        uc3     device 'P16R4';

        Clk,Clr,Dir,OE  pin 1,2,3,11;
        Q3,Q2,Q1,Q0     pin 14,15,16,17;

        Ck,X,Z  = .C. , .X., .Z.;

" Counter States
        S0 = ^b1111;    S4 = ^b1011;    S8 = ^b0111;
        S1 = ^b1110;    S5 = ^b1010;    S9 = ^b0110;
        S2 = ^b1101;    S6 = ^b1001;
        S3 = ^b1100;    S7 = ^b1000;

" Counter modes
        Mode    = [Clr,Dir];
        Up      = [ 1 , 1 ];
        Down    = [ 1 , 0 ];
        Clear   = [ 0 , X ];

test_vectors (  [Clk ,OE, Mode ] -> [Q3,Q2,Q1,Q0])
                [ Ck , 0, Clear] ->    S0;
                [ Ck , 0, Up   ] ->    S1;
                [ Ck , 0, Up   ] ->    S2;
                [ Ck , 0, Up   ] ->    S3;
                [ Ck , 0, Up   ] ->    S4;
                [ Ck , 0, Up   ] ->    S5;
                [ Ck , 0, Up   ] ->    S6;
                [ Ck , 0, Up   ] ->    S7;
                [ Ck , 0, Up   ] ->    S8;
                [ Ck , 0, Up   ] ->    S9;
                [ Ck , 0, Up   ] ->    S0;
                [ Ck , 0, Up   ] ->    S1;
                [ Ck , 0, Down ] ->    S0;
                [ Ck , 0, Down ] ->    S9;
                [ Ck , 1, Down ] ->     Z;
                [ Ck , 0, Down ] ->    S7;
                [ Ck , 0, Clear] ->    S0;
@page
```

Listing 2–5. Source File: Decimal Up/Down Counter With Synchronous Clear

Listing continued on next page.

```
state_diagram [Q3,Q2,Q1,Q0]

        State S0:       case (Mode == Up)    : S1;
                             (Mode == Down)  : S9;
                             (Mode == Clear) : S0;
                        endcase;

        State S1:       case (Mode == Up)    : S2;
                             (Mode == Down)  : S0;
                             (Mode == Clear) : S0;
                        endcase;

        State S2:       case (Mode == Up)    : S3;
                             (Mode == Down)  : S1;
                             (Mode == Clear) : S0;
                        endcase;

        State S3:       case (Mode == Up)    : S4;
                             (Mode == Down)  : S2;
                             (Mode == Clear) : S0;
                        endcase;

        State S4:       case (Mode == Up)    : S5;
                             (Mode == Down)  : S3;
                             (Mode == Clear) : S0;
                        endcase;

        State S5:       case (Mode == Up)    : S6;
                             (Mode == Down)  : S4;
                             (Mode == Clear) : S0;
                        endcase;

        State S6:       case (Mode == Up)    : S7;
                             (Mode == Down)  : S5;
                             (Mode == Clear) : S0;
                        endcase;

        State S7:       case (Mode == Up)    : S8;
                             (Mode == Down)  : S6;
                             (Mode == Clear) : S0;
                        endcase;

        State S8:       case (Mode == Up)    : S9;
                             (Mode == Down)  : S7;
                             (Mode == Clear) : S0;
                        endcase;

        State S9:       case (Mode == Up)    : S0;
                             (Mode == Down)  : S8;
                             (Mode == Clear) : S0;
                        endcase;
end count10
```

Listing 2–5. *(Continued)*

8-BIT BARREL SHIFTER (EQUATIONS/P20R8) 2.6

This design for an 8-bit barrel shifter includes a shift amount selector, an output control, and a device enable. The design is specified with one Boolean equation for a P20R8.

Design Specification

Figure 2-10 shows a block diagram for the barrel shifter. The shifter has eight inputs (*D0–D7*), eight outputs (Q0–Q7), three select lines (*I0–I2*), a clock (*Clk*), an output control (*OC*), and an enable (*E*). On each clock pulse when *E* is high, the outputs show the inputs shifted by n bits to the right, where n is specified by the select lines. The bit shifted out of the barrel shifter on the right is shifted in on the left, actually performing a rotate. When *E* is low, the shifter outputs are preset to 1.

The output control, when high, sets all outputs to high impedance, without affecting the shift. This means that if a shift is selected while the output control is high, the shift still occurs, but it is not seen at the outputs. If the *OC* is then set low, the shifted data will appear on the outputs.

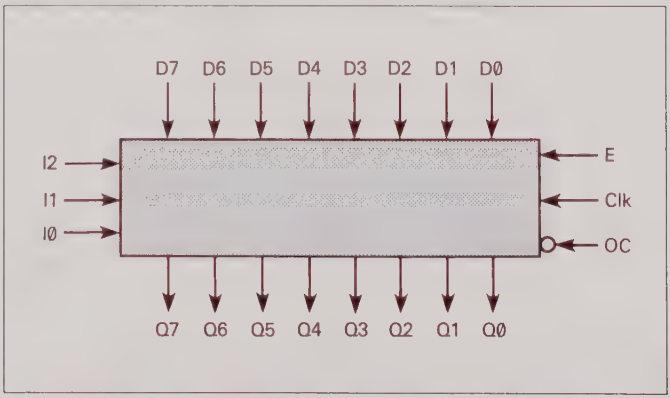


Figure 2-10. Block Diagram: 8-Bit Barrel Shifter

Design Method

Figure 2–11 and listing 2–6 show the simplified block diagram and the source file listing for this design. Pins are assigned so that the shifter outputs are associated with the registered outputs on the PAL. The inputs, outputs, and select lines are then assigned to sets to simplify notation.

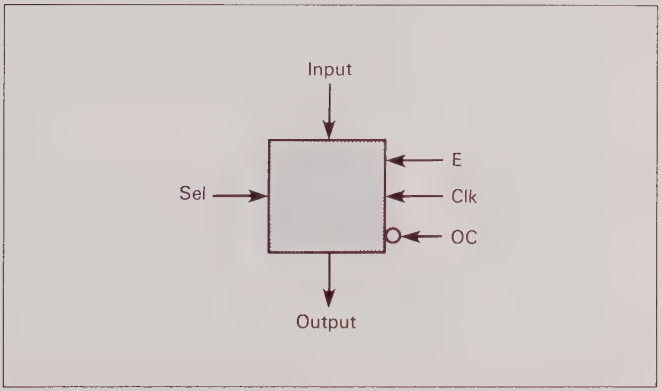


Figure 2–11. Simplified Block Diagram: 8–Bit Barrel Shifter

One Boolean equation is used to describe the entire function of the barrel shifter. The equation is expressed in sum of products form and assigns a value to the output set. Each product in the equation corresponds to one of the possible shifts and defines the outputs for that shift. Thus, the product term,

```
(Sel == 0) & ![D7,D6,D5,D4,D3,D2,D1,D0]
```

defines that for a shift of 0, the inputs are transferred without a shift directly to the outputs. Similarly, the product term,

```
(Sel == 5) & ![D4,D3,D2,D1,D0,D7,D6,D5]
```

defines that for a shift of 5, output *Q7* gets the value of input *D4*, *Q6* gets the value of *D3*, and so on, corresponding to the correct shift of 5 places. Notice that the low–order input bits have been "wrapped around," shifted out of the right side and into the left side.

Sel can have only one value at a time, thus only one of the "Sel == " relational statements can be true at a given time, and only one of the product terms contributes to the sum of products. The OR of all the product terms is ANDed with the enable *E* so that when *E* is low, all the outputs are preset to 1.

Both the output set on the left side of the equation and the inputs on the right side of the equation are expressed as negative logic, which, in effect, gives active high logic. This is done to compensate for the P20R8's inverted outputs. The inverse of the inputs is available on the PAL.

Test Vectors

The test vectors are written to check the shift, enable and output control functions of the barrel shifter. To test the shift function, *OC* is set low, *E* is set high, the clock is applied and different *Sel* values are chosen. The shift is first tested with one input bit set high and the rest of the inputs set low. Then, as a further test, one input bit is set low and the remaining inputs are set high. In both cases, the bits are shifted through one full cycle plus one additional shift so that the wrap-around shift from Q0 to Q7 is tested.

The preset is tested by setting *E* low; all inputs should go high. The output control is tested by setting *OC* high; all outputs should go to high impedance. The single Z in the last test vector expands to cover all outputs. That is, the Z becomes [Z,Z,Z,Z,Z,Z,Z,Z] to cover all eight outputs.

```
module  barrel

title '8-bit barrel shifter
Gerrit Barrere    Data I/O Corp  Redmond WA   24 Feb 1984'

        P7095   device  'P20R8';

        D7,D6,D5,D4,D3,D2,D1,D0          Pin 2,3,4,5,6,7,8,9;
        Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0          Pin 15,16,17,18,19,20,21,22;
        Clk,OC,E                         Pin 1,13,23;
        I2,I1,I0                         Pin 10,11,14;

        Input           = [D7,D6,D5,D4,D3,D2,D1,D0];
        Output          = [Q7,Q6,Q5,Q4,Q3,Q2,Q1,Q0];
        Sel             = [I2,I1,I0];

        H,L,C,Z         = 1,0,.C.,.Z.;

equations

        !Output := E & (  (Sel == 0) & ![D7,D6,D5,D4,D3,D2,D1,D0]
                        # (Sel == 1) & ![D0,D7,D6,D5,D4,D3,D2,D1]
                        # (Sel == 2) & ![D1,D0,D7,D6,D5,D4,D3,D2]
                        # (Sel == 3) & ![D2,D1,D0,D7,D6,D5,D4,D3]
                        # (Sel == 4) & ![D3,D2,D1,D0,D7,D6,D5,D4]
                        # (Sel == 5) & ![D4,D3,D2,D1,D0,D7,D6,D5]
                        # (Sel == 6) & ![D5,D4,D3,D2,D1,D0,D7,D6]
                        # (Sel == 7) & ![D6,D5,D4,D3,D2,D1,D0,D7]) ;

test_vectors
        ([Clk,OC, E, Sel,   Input]    ->   Output)
         [ C,  L, H,  0, ^b10000000]  -> ^b10000000;    " Shift 0
         [ C,  L, H,  1, ^b10000000]  -> ^b01000000;    " Shift 1
         [ C,  L, H,  2, ^b10000000]  -> ^b00100000;    " Shift 2
         [ C,  L, H,  3, ^b10000000]  -> ^b00010000;    " Shift 3
         [ C,  L, H,  4, ^b10000000]  -> ^b00001000;    " Shift 4
         [ C,  L, H,  5, ^b10000000]  -> ^b00000100;    " Shift 5
         [ C,  L, H,  6, ^b10000000]  -> ^b00000010;    " Shift 6
         [ C,  L, H,  7, ^b10000000]  -> ^b00000001;    " Shift 7

         [ C,  L, H,  0, ^b01111111]  -> ^b01111111;    " Shift 0
         [ C,  L, H,  1, ^b01111111]  -> ^b10111111;    " Shift 1
         [ C,  L, H,  2, ^b01111111]  -> ^b11011111;    " Shift 2
         [ C,  L, H,  3, ^b01111111]  -> ^b11101111;    " Shift 3
         [ C,  L, H,  4, ^b01111111]  -> ^b11110111;    " Shift 4
         [ C,  L, H,  5, ^b01111111]  -> ^b11111011;    " Shift 5
         [ C,  L, H,  6, ^b01111111]  -> ^b11111101;    " Shift 6
         [ C,  L, H,  7, ^b01111111]  -> ^b11111110;    " Shift 7

         [ C,  L, H,  1, ^b00000001]  -> ^b10000000;    " Shift 1/Wrap
         [ C,  L, H,  1, ^b11111110]  -> ^b01111111;    " Shift 1/Wrap
         [ C,  L, L,  0, ^b00000000]  -> ^b11111111;    " Preset

         [ C,  H, H,  0, ^b00000000]  ->  Z             " Test High Z
end
```

Listing 2-6. Source File: 8-Bit Barrel Shifter

7-SEGMENT DISPLAY DECODER (TRUTH TABLE/RA5P8) 2.7

This display decoder decodes a four-bit binary number to display the decimal equivalent on a seven segment LED display. The design incorporates a truth table.

Design Specification

Figure 2-12 shows a block diagram for the design of a 7-segment display decoder and a drawing of the display with each of the seven segments labeled to correspond to the decoder outputs. To light up any one of the segments, the corresponding line must be driven low. Four input lines *D0-D3* are decoded to drive the correct output lines. The outputs are named *a, b, c, d, e, f,* and *g* corresponding to the display segments. All outputs are active low. An enable, *ena*, is provided. When *ena* is low, the decoder is enabled; when *ena* is high, all outputs are driven to high impedance.

Design Method

Figure 2-13 and listing 2-7 show the simplified block diagram and the source file for the ABEL™ implementation of the display decoder. The FLAG statement is used to make sure that the programmer load file is in the Motorola Exorciser format. The binary inputs and the decoded outputs are grouped into the sets *bcd* and *led* to simplify notation. The constants *ON* and *OFF* are declared so that the design can be described in terms of turning a segment on or off. To turn a segment on, the appropriate line must be driven low, thus we declare *ON* as 0 and *OFF* as 1.

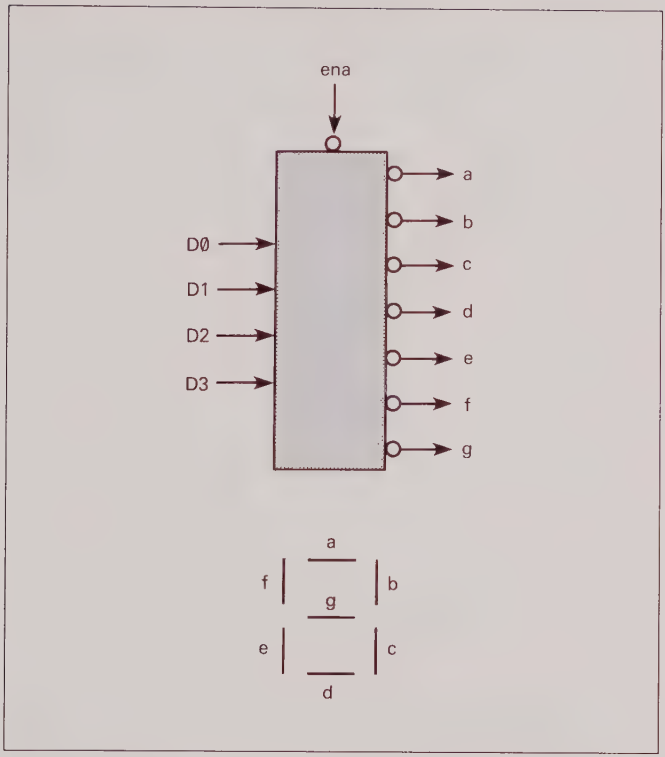


Figure 2–12. Block Diagram: 7–Segment Display Decoder

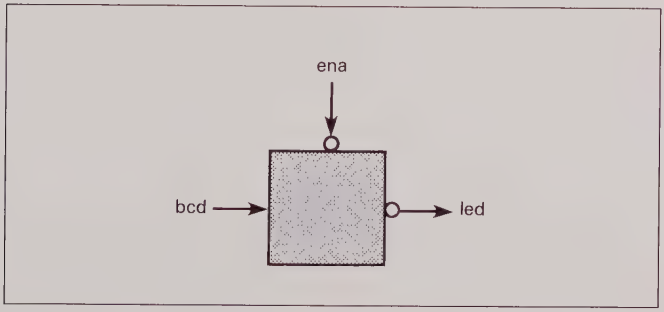


Figure 2–13. Simplified Block Diagram: 7–Segment Display Decoder

The design is described in two sections, an equations section and a truth table section. The decoding function is described with a truth table that specifies the outputs required for each combination of inputs. The truth table header names the inputs and outputs. In this example, the inputs are contained in the set named *bcd* and the outputs are in *led*. The body of the truth table defines the input to output function. Because the design decodes a number to a seven segment display, values for *bcd* are expressed as decimal numbers, and values for *led* are expressed with the constants *ON* and *OFF* that were defined in the declarations section of the source file. This makes the truth table easy to read and understand; the incoming value is a number and the outputs are on and off signals to the LED.

The input and output values could have just as easily been described in another form. Take for example the line in the truth table:

```
5 -> [ ON, OFF, ON, ON, OFF, ON, ON]
```

This could have been written in the equivalent form:

```
[ 0, 1, 0, 1 ] -> 36
```

In this second form, 5 was simply expressed as a set containing binary values, and the LED set was converted to decimal. (Remember that *ON* was defined as 0 and *OFF* was defined as 1.) Either of the two forms is valid, but the first is more appropriate for this design. The first form can be read as, "the number five turns on the first segment, turns off the second, . . ." whereas the second form cannot be so easily translated into terms meaningful for this design.

Test Vectors

The test vectors for this design test the decoder outputs for the ten valid combinations of input bits. The enable is also tested by setting *ena* high for the different combinations. All outputs should be at high impedance whenever *ena* is high.

```
module  bcd7rom flag '-d82'
title 'seven segment display decoder
Data I/O Corp    Redmond WA   16 Mar 1984'
"          a
"         ---
"       f| g |b        BCD-to-seven-segment decoder similar to the 7449
"         ---
"       e| d |c              segment identification
"         ---
"  ---          ---      ---          ---      ---      ---      ---
" |   |  |      |   |    |   |  |    |   |    |   |    |   |    |   |
"  ---          ---      ---          ---      ---      ---      ---
" |   |  |      |   |    |   |  |    |   |    |   |    |   |    |   |
"  ---          ---      ---          ---      ---      ---      ---

        U6      device  'RA5P8';

        D3,D2,D1,D0     pin 10,11,12,13;
        a,b,c,d,e,f,g   pin 1,2,3,4,5,6,7;
        ena             pin 15;

        bcd     = [D3,D2,D1,D0];
        led     = [a,b,c,d,e,f,g];

        ON,OFF  = 0,1;                          " for common anode LEDs
        L,H,X,Z = 0,1,.X.,.Z.;

truth_table (bcd  -> led)
"  input                    outputs
"              a    b    c    d    e    f    g
     0  -> [ ON,  ON,  ON,  ON,  ON,  ON, OFF];
     1  -> [OFF,  ON,  ON, OFF, OFF, OFF, OFF];
     2  -> [ ON,  ON, OFF,  ON,  ON, OFF,  ON];
     3  -> [ ON,  ON,  ON,  ON, OFF, OFF,  ON];
     4  -> [OFF,  ON,  ON, OFF, OFF,  ON,  ON];
     5  -> [ ON, OFF,  ON,  ON, OFF,  ON,  ON];
     6  -> [ ON, OFF,  ON,  ON,  ON,  ON,  ON];
     7  -> [ ON,  ON,  ON, OFF, OFF, OFF, OFF];
     8  -> [ ON,  ON,  ON,  ON,  ON,  ON,  ON];
     9  -> [ ON,  ON,  ON,  ON, OFF,  ON,  ON];

test_vectors ([ena,bcd]  -> led)
" inputs                 outputs
"              a    b    c    d    e    f    g
  [L,1]  -> [OFF,  ON,  ON, OFF, OFF, OFF, OFF];
  [L,2]  -> [ ON,  ON, OFF,  ON,  ON, OFF,  ON];
  [L,3]  -> [ ON,  ON,  ON,  ON, OFF, OFF,  ON];
  [L,4]  -> [OFF,  ON,  ON, OFF, OFF,  ON,  ON];
  [L,5]  -> [ ON, OFF,  ON,  ON, OFF,  ON,  ON];
  [L,6]  -> [ ON, OFF,  ON,  ON,  ON,  ON,  ON];
  [L,7]  -> [ ON,  ON,  ON, OFF, OFF, OFF, OFF];
  [L,8]  -> [ ON,  ON,  ON,  ON,  ON,  ON,  ON];
  [L,9]  -> [ ON,  ON,  ON,  ON, OFF,  ON,  ON];
  [L,0]  -> [ ON,  ON,  ON,  ON,  ON,  ON, OFF];
  [H,5]  -> [  Z,   Z,   Z,   Z,   Z,   Z,   Z];
  [H,9]  -> [  Z,   Z,   Z,   Z,   Z,   Z,   Z];
  [H,0]  -> [  Z,   Z,   Z,   Z,   Z,   Z,   Z];
end     bcd7rom
```

Listing 2–7. Source File: 7–Segment Display Decoder

4-BIT COMPARATOR (MACROS,EQUATIONS/P16HD8) 2.8

This is a design for a 4-bit comparator that provides an output for "equal to", "less than", "not equal to", and "greater than", as well as intermediate outputs. The design is implemented with macros and Boolean equations. Another implementation of this same design in a PROM can be found in the source file named COMP4ROM.ABL on the Design Examples disk or tape.

Design Specification

The comparator, as shown in figure 2-14, compares the values of two four-bit inputs (*A0–A3* and *B0–B3*) and determines whether *A* is equal to, not equal to, less than, or greater than *B*. The result of the comparison is shown on the output lines, *A_EQ_B*, *A_GT_B*, *A_NE_B*, and *A_LT_B*. Intermediate outputs used in calculation of the result are also provided on the lines *E0–E3*.

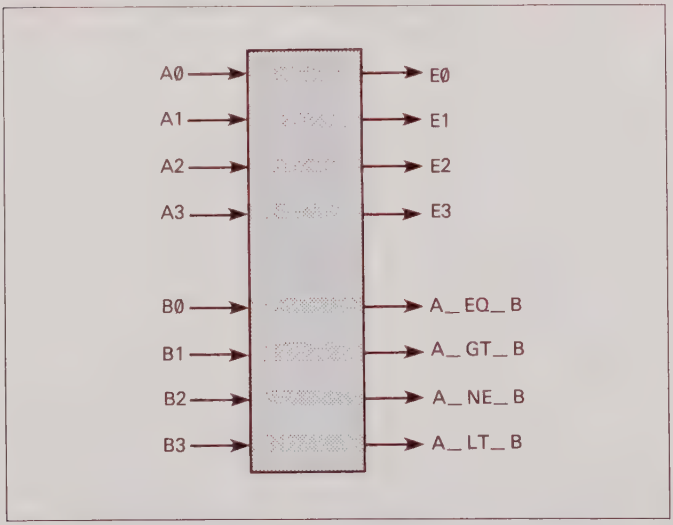


Figure 2-14. Block Diagram: 4-Bit Comparator

Design Method

Figure 2–15 and listing 2–8 show the simplified block diagram and source file listing for the comparator. The inputs *A0–A3*, *B0–B3*, and the outputs *E0–E3* are grouped into the sets *A*, *B*, and *E*. *YES* and *NO* are defined as 1 and 0, to be used in the test vectors.

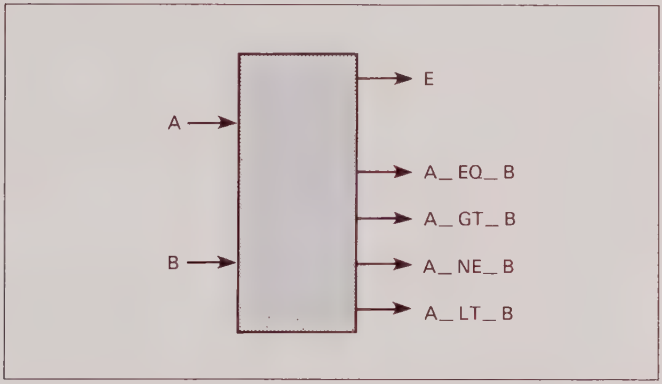


Figure 2–15. Simplified Block Diagram: 4-bit Comparator

Design of this comparator is not as straightforward as might be expected. ABEL™ operators include greater than, less than, equal to, and not equal to, and it would seem that these operators could be used directly to produce the desired function. However, such an implementation is not possible with the chosen device and the size of the comparator. Direct application of the relational operators for a problem of this size may cause several problems. First the comparison of sets of signals produces a large number of exclusive-OR operators that take a significant amount of time and memory to reduce (they may take too much memory). Also, after the equations are reduced, there may be too many product terms for the chosen device. In either case, the ABEL™ language processor would detect and report the error.

```
module comp4
title '4-bit look-ahead comparator
Charles Olivier & Mary Bailey    Data I/O Corp.   24 Feb 1984'

        IC1     device 'P16HD8';

        A3,A2,A1,A0     pin     1,2,3,4;
        B3,B2,B1,B0     pin     5,6,7,8;
        E3,E2,E1,E0     pin     14,15,16,17;
        A_NE_B,A_EQ_B   pin     12,13;
        A_GT_B,A_LT_B   pin     18,19;

        No,Yes  = 0,1;
        A       = [A3,A2,A1,A0];
        B       = [B3,B2,B1,B0];
        E       = [E3,E2,E1,E0];
        C0      macro [(A0>B0)];
        C1      macro [(A1>B1) # E1 & (C0)];
        C2      macro [(A2>B2) # E2 & (C1)];

equations
        E       = A !$ B;       "intermediate An = Bn

        A_EQ_B  = E3 & E2 & E1 & E0;

        A_NE_B  = !A_EQ_B;

        A_GT_B  = (A3>B3) # E3 & (C2);

        A_LT_B  = !A_GT_B & !A_EQ_B;

test_vectors  'test for A = B'
      ([ A, B] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
        [ 0, 0] -> [ Yes  ,   No  ,   No  ,   No  ];
        [ 1, 1] -> [ Yes  ,   No  ,   No  ,   No  ];
        [ 2, 2] -> [ Yes  ,   No  ,   No  ,   No  ];
        [ 5, 5] -> [ Yes  ,   No  ,   No  ,   No  ];
        [ 8, 8] -> [ Yes  ,   No  ,   No  ,   No  ];
        [10,10] -> [ Yes  ,   No  ,   No  ,   No  ];
        [15,15] -> [ Yes  ,   No  ,   No  ,   No  ];

test_vectors  'test for A > B'
      ([ A, B] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
        [ 1, 0] -> [  No  ,  Yes  ,   No  ,  Yes  ];
        [ 2, 1] -> [  No  ,  Yes  ,   No  ,  Yes  ];
        [ 4, 3] -> [  No  ,  Yes  ,   No  ,  Yes  ];
        [ 8, 7] -> [  No  ,  Yes  ,   No  ,  Yes  ];
        [15,14] -> [  No  ,  Yes  ,   No  ,  Yes  ];
        [ 6, 2] -> [  No  ,  Yes  ,   No  ,  Yes  ];
        [ 5, 0] -> [  No  ,  Yes  ,   No  ,  Yes  ];

test_vectors  'test for A < B'
      ([ A, B] -> [A_EQ_B, A_GT_B, A_LT_B, A_NE_B])
        [ 3, 9] -> [  No  ,   No  ,  Yes  ,  Yes  ];
        [14,15] -> [  No  ,   No  ,  Yes  ,  Yes  ];
        [ 7, 8] -> [  No  ,   No  ,  Yes  ,  Yes  ];
        [ 3, 4] -> [  No  ,   No  ,  Yes  ,  Yes  ];
        [ 2, 8] -> [  No  ,   No  ,  Yes  ,  Yes  ];
end comp4
```

Listing 2-8. Source File: 4-Bit Comparator

NOTE: Choosing another device can eliminate such problems. The source file named COMP4ROM.ABL on the Design Examples disk or tape shows this same design implemented on a 256 X 4 ROM, which, with 256 product terms, can accommodate a straightforward implementation.

Thus, another approach is taken in this design. The comparison is made on a bit by bit basis and the results of the bit comparisons are combined to create the full comparator function. The algorithm used is as follows:

1. **Determine bit by bit equality.** The exclusive-NOR of sets *A* and *B* is assigned to set *E*. The exclusive-NOR causes a bit by bit comparison for equality between the two sets. Bit *E0* will be a 1 if bits *A0* and *B0* are equal, bit *E1* will be a 1 if bits *A1* and *B1* are equal, and so on with bits 2 and 3.

2. **Check for set equality.** *A* equals *B* if all bits of *E* are 1's.

3. **Check for A > B.** The simplest case for this condition is that the high order bit of *A* is a one while the high-order bit of *B* is a zero. Otherwise, *A* can be greater than *B* only if their high-order bits are equal, and the remaining three lower order bits of *A* are greater than the remaining bits in *B*. The check on each of the three remaining bits is identical to the check performed on the high-order bits.

4. **Check for less than and not equal to.** These two conditions are determined by the results of steps 2 and 3.

For example, let *A* equal binary 1010 (decimal 10) and *B* equal binary 1001 (decimal 9). The algorithm is applied as follows:

Step 1: E equals XNOR of A and B

```
E = A !$ B;
[E3,E2,E1,E0] = [A3,A2,A1,A0] !$ [B3,B2,B1,B0];
[E3,E2,E1,E0] = [1,0,1,0] !$ [1,0,0,1];
[E3,E2,E1,E0] = [1,1,0,0];
E3 = 1; E2 = 1; E1 = 0; E0 = 0;
```

Step 2: A does not equal B because E is not all 1's. A__EQ__B = 0

Step 3 The simple case of A3 = 1, B3 = 0 does not hold, so a bit by bit check must be performed. A can only be greater than B if A3 and B3 are equal (E3 = 1) and the three remaining bits of A are greater than the three remaining bits of B. A3 and B3 are equal. The check on the remaining bits begins with A2 and B2. The simple case of A2 = 1, B2 = 0 is not true, but A2 and B2 are both 0's so the two remaining bits must be checked. For bits A1 and B1, the simple case, A1 = 1, B1 = 0, is true, so the check can be terminated—A is greater than B.

Step 4: A is not equal to B as determined in step 2, so A__NE__B is set to 1. A is greater than B, thus A cannot be less than B, and A__LT__B is set to 0.

This somewhat complicated algorithm is described quite easily with ABEL™. Steps 1 and 2 of the algorithm are implemented directly with Boolean equations. The more involved Step 3 is simplified by using three macros. *A__GT__B* is formed by an equation that checks the high order bits against the simple *A3* = 1, *B3* = 0 condition, and then checks the next lowest order bit by invoking the macro *C2*. *C2* performs the same check for bit 2 and invokes *C1* to check bit 1. *C1* in turn invokes C0 to check bit 0.

Macros are expanded when the language processor is run, so the line,

```
A__GT__B = (A3>B3) # E3 & (C2);
```

becomes,

```
A__GT__B = (A3>B3)# E3 & ((A2>B2) # E2 & ((A1>B1)
           # E1 & (A0>B0)));
```

which fully implements step 3 of the algorithm. The macros simplify notation in the source file and help to show the function as the bit by bit check that it is.

Test Vectors

Three separate test vectors sections are written to test three of the four possible conditions. (The fourth and untested condition of NOT EQUAL TO is simply the inverse of EQUAL TO.) Each test vectors table includes a test vector message that helps make output from the documentation generator (DOCUMENT) and the simulator (SIMULATE) easier to read. The three tested conditions are not mutually exclusive, so one or more of them can be met by a given *A* and *B*. In the test vectors table, the constants *YES* and *NO* are used rather than 1 and 0, just for ease of reading. *YES* and *NO* are declared in the declaration section of the source file.

BI-DIRECTIONAL TRI-STATE BUFFER (IFL DEVICE) 2.9

A four-bit bidirectional buffer with tri-state outputs is presented here. The design is implemented in an F82S153 IFL with bidirectional inputs/outputs and programmable output polarity. Simple Boolean equations are used to describe the function.

Design Specification 2.9.1

Figure 2-16 shows a block diagram for this four-bit buffer. Signals *A0-A3* and *B0-B3* function both as inputs and outputs depending on the value on the select lines, *S0-S1*. When the select value (the value on the select lines) is 1, *A0-A3* are enabled as outputs. When the select value is 2, *B0-B3* are enabled as outputs. (The choice of 1 and 2 for select values is arbitrary.) For any other values of the select lines, both the A and B outputs are at high impedance. Output polarity for this design is positive.

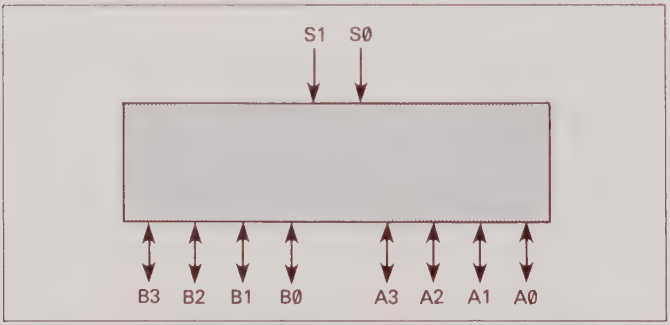


Figure 2-16. Block Diagram: Bidirectional Tri-State Buffer

Design Method 2.9.2

A simplified block diagram for the buffer is shown in figure 2–17. The A and B inputs/outputs are grouped into two sets, *A* and *B*. The select lines are grouped into the *select* set. Listing 2–9 shows the source file that describes the design.

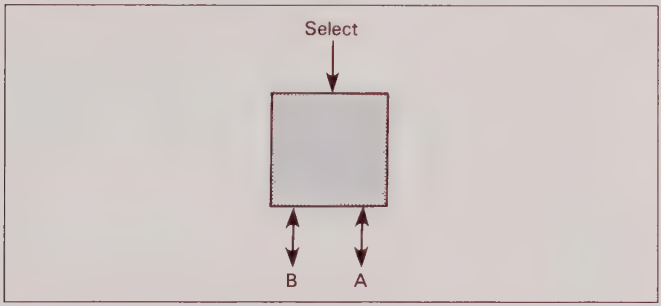


Figure 2–17. Simplified Block Diagram: Tri-State Buffer

High-impedance and don't-care values are declared to simplify notation in the source file. The equations section describes the full function of the design. What appear to be unresolvable equations are written for *A* and *B*, with both sets appearing as inputs and outputs. The enable equations, however, enable only one set at a time as outputs; the other set functions as inputs to the buffer.

Test vectors are written to test the buffer when either set is selected as the output set and for the case when neither is selected. The test vectors are written in terms of the previously declared sets so that the element values do not need to be listed separately.

```
module tsbuffer
title 'bi-directional three state buffer
Brenda French   FutureNet -  Data I/O Corp   8 Nov 1985'

        TSB1    device 'F82S153';

        S1,S0           Pin 1,2;            Select = [S1,S0];
        A3,A2,A1,A0     Pin 12,13,14,15;    A      = [A3,A2,A1,A0];
        B3,B2,B1,B0     Pin 16,17,18,19;    B      = [B3,B2,B1,B0];

        X,Z     = .X., .Z.;

equations
        A = B,
        B = A,

        Enable A = (Select == 1),
        Enable B = (Select == 2),

test vectors
        ([Select,   A,  B]-> [  A,  B])
         [  0   ,  0,  0]-> [  Z,  Z];
         [  0   , 15, 15]-> [  Z,  Z];

         [  1   ,  X,  5]-> [  5,  X];
         [  1   ,  X, 10]-> [ 10,  X],

         [  2   ,  5,  X]-> [  X,  5],
         [  2   , 10,  X]-> [  X, 10];

         [  3   ,  0,  0]-> [  Z,  Z],
         [  3   , 15, 15]-> [  Z,  Z];
end
```

Listing 2-9. Source File: Tri-State Bi-Directional Buffer

DESIGN CONSIDERATIONS 3

General information and considerations that you may find useful as you design logic with ABEL™ are collected into this section. The section is organized by topic, and examples are provided where appropriate. The section is divided into five subsections as follows:

3.1 Using State Machines
3.2 Multiple Assignments to the Same Signal
3.3 Hazard–Free Logic and REDUCE
3.4 Passing Arguments From the Command Line
3.5 Effect of Equation Polarity on Reduction Speed

USING STATE MACHINES 3.1

State machines are one of the more powerful features of ABEL™, but some care should be taken in organizing the state diagram and in ordering the states within it. This section describes some of the steps you can take to make state diagrams easy to read and maintain and to avoid some potential problems. The most common problem encountered with state machines is that too many product terms are created for the chosen device. This problem arises because state machines often have many different states and complex state transitions. The topics discussed in the following subsections will help you avoid this problem by reducing the number of required product terms.

The subsections are:

3.1.1 Use Identifiers Rather Than Numbers for States
3.1.2 "Power On" Register States
3.1.3 Design With Registers, not With Outputs
3.1.4 Unsatisfied Transition Conditions Clear the State Register
3.1.5 Number Adjacent States for One Bit Changes
3.1.6 Use State Register Outputs to Identify States
3.1.7 Combine the Use of Equations and State Diagrams

Use Identifiers Rather Than Numbers for States 3.1.1

As you develop a state diagram, you need to label the various states and state transitions. It is best to label the states with identifiers that have been assigned constant values rather than labeling the states directly with numbers. This allows you to change the state transitions easily or to change the state register values associated with each state.

A state machine has different "states" that describe the outputs and transitions of the machine at any given point. Typically, each state is given a name, and the state machine is described in terms of transitions from one state to another. In a real device, such a state machine is implemented with registers that contain enough bits to assign a unique number to each state. The states are actually bit values in the register, and these bit values are used along with other signals to determine state transitions.

In writing a state diagram with ABEL™, you should follow the same design procedure of first describing the machine with names for the states, and then assigning state register bit values to the state names.

For an example, see listing 3–1, which lists the source file for a four–bit decimal counter named COUNT10. This counter is a slight variation on the design discussed fully in section 2.5. In the state diagram, identifiers (*S0, S1, S2* . . .) are used to specify the states. These identifiers are assigned a constant binary value in the declaration section of the source file that corresponds to the bit values in the state register for each state. Thus, the counter was designed conceptually, with transitions from state to state occurring in a logical sequence based on the state names. *S0* goes to *S1* to *S2* and so on as the count progresses from 0 to 1 to 2. But, in the constant declarations, the bit values are not all 0's for state 0, but 1's. The declared constant values take into account the actual implementation of the state machine in the device: The P16R4 PAL has active–low outputs.

```
module count10a flag '-r3'
title 'decimal Up/Down counter with synchronous Clear
Michael Holley      Data I/O Corp    14 Mar 1984'

        uc3a    device 'P16R4';

        Clk,Clr,Dir,OE  pin 1,2,3,11;
        Q3,Q2,Q1,Q0     pin 14,15,16,17;

        Ck,X,Z  = .C. , .X., .Z.;

" Counter States
        S0 = ^b1111;    S4 = ^b1011;    S8  = ^b0111;
        S1 = ^b1110;    S5 = ^b1010;    S9  = ^b0110;
        S2 = ^b1101;    S6 = ^b1001;
        S3 = ^b1100;    S7 = ^b1000;

" Counter modes
        Mode    = [Clr,Dir];
        Up      = [ 1 , 1 ];
        Down    = [ 1 , 0 ];
        Clear   = [ 0 , X ];

test_vectors (  [Clk ,OE, Mode ] -> [Q3,Q2,Q1,Q0])
                [ Ck , 0, Clear] ->    S0;
                [ Ck , 0, Up   ] ->    S1;
                [ Ck , 0, Up   ] ->    S2;
                [ Ck , 0, Up   ] ->    S3;
                [ Ck , 0, Up   ] ->    S4;
                [ Ck , 0, Up   ] ->    S5;
                [ Ck , 0, Up   ] ->    S6;
                [ Ck , 0, Up   ] ->    S7;
                [ Ck , 0, Up   ] ->    S8;
                [ Ck , 0, Up   ] ->    S9;
                [ Ck , 0, Up   ] ->    S0;
                [ Ck , 0, Up   ] ->    S1;
                [ Ck , 0, Down ] ->    S0;
                [ Ck , 0, Down ] ->    S9;
                [ Ck , 1, Down ] ->    Z ;
                [ Ck , 0, Down ] ->    S7;
                [ Ck , 0, Clear] ->    S0;
@page
```

Listing 3-1. Source File: Decimal Counter

Listing continued on next page.

```
equations

        ![Q3,Q2,Q1,Q0] := (Mode == Clear) & ![1,1,1,1];

state_diagram [Q3,Q2,Q1,Q0]

        State S0:       case (Mode == Up)     : S1;
                              (Mode == Down)   : S9.
                        endcase.

        State S1:       case (Mode == Up)     : S2;
                              (Mode == Down)   : S0;
                        endcase.

        State S2:       case (Mode == Up)     : S3;
                              (Mode == Down)   : S1;
                        endcase.

        State S3:       case (Mode == Up)     : S4;
                              (Mode == Down)   : S2;
                        endcase;

        State S4:       case (Mode == Up)     : S5;
                              (Mode == Down)   : S3;
                        endcase;

        State S5:       case (Mode == Up)     : S6;
                              (Mode == Down)   : S4;
                        endcase;

        State S6:       case (Mode == Up)     : S7;
                              (Mode == Down)   : S5;
                        endcase;

        State S7:       case (Mode == Up)     : S8;
                              (Mode == Down)   : S6;
                        endcase;

        State S8:       case (Mode == Up)     : S9;
                              (Mode == Down)   : S7;
                        endcase;

        State S9:       case (Mode == Up)     : S0;
                              (Mode == Down)   : S8;
                        endcase;

end count10a
```

Listing 3–1. *(Continued)*

"Power On" Register States 3.1.2

Your state machine design might depend on a specific starting state, and, therefore, you must know what the "power–on" state of a device's register is. If the register powers on in a state that is not defined in the state diagram description of the state machine, the next state is undefined and unknown unless your design specifically accounts for this situation. Therefore, you should account for the power–on state of the device you are using or make sure that your design goes to a known state at power–on time.

Design With the Registers, Not With the Outputs 3.1.3

With devices featuring programmable polarity on the outputs, you must be sure to design your state machine with the register outputs, not the device outputs, in mind. This is important because all the sets and clears are on the registers rather than on the programmable outputs and because you are concerned with the states themselves rather than with the outputs. The final outputs can be handled separately.

Unsatisfied Transition Conditions Clear the State Register 3.1.4

For each state described in a state diagram, you specify the transitions to the next state and the conditions that influence those transitions. For devices with D-type flip-flops, if none of the stated conditions is met, the state register is cleared to all 0's and on the next clock pulse, the state machine goes to the state corresponding to a cleared state register. (This state is called the "cleared-register state.") This can either be used to your advantage, or cause problems, depending on your design.

You can use this fact to eliminate some conditions in your state diagram and some product terms in the converted design. Rather than having a condition for a transition to the cleared-register state, that transition can be left implicit. If none of the other transition conditions is met, the machine will go to the cleared state. This same fact can cause problems if the cleared state is undefined in the state diagram, because if the transition conditions are not met for any state, the machine will go to the undefined cleared state, and the next state after that is unknown.

In general, this means that you should always have a state assigned to the cleared-register state. Or, you must define every possible condition so that some condition is always met for each state. But, the automatic transition to the cleared-register state can also be used to your advantage by eliminating product terms and explicit definitions of transitions.

Number Adjacent States for One-Bit Changes 3.1.5

The number of product terms produced by a state diagram can be reduced greatly by a careful choice of state register bit values. Your state machine should be described with symbolic names for the states, as described in section 3.1.1. Then if you assign the numeric constants to these names such that the state register bits change by only one bit at a time as the state machine goes from state to state, the number of product terms required to describe the state transitions is reduced.

For example, take the states, *A, B, C,* and *D,* which go from one state to the other alphabetically. The simplest choice of bit values for the state register is a simple numeric sequence. The simplest choice is not, however, the most efficient. Take, for example, the following simple and preferred choices for bit value assignments:

State	Simple Bit Values	Preferred Bit Values
A	00	00
B	01	01
C	10	11
D	11	10

Notice that the preferred bit values cause a change of only one bit as the machine moves from state B to C, whereas the simple choices for the bit values cause a change in both bit values for the same transition. The preferred bit values will produce fewer product terms.

One specialized case occurs when the language processor reports that too many product terms were produced for one of the state register bits. If this occurs, you should reorganize the bit values so that as the state machine moves from state to state, the bit for which there are too many terms changes in value as few times as possible.

Obviously, the choice of the optimum bit values for specific states can require some tradeoffs; you may have to optimize for one bit, and, in the process, increase the value changes for another. The overall object should be to eliminate as many product terms as is necessary to fit the design into the device.

Use State Register Outputs To Identify States 3.1.6

Sometimes it is necessary to identify specific states of a state machine and signal on an output that the machine is in one of these specific states. Equations and outputs can be saved if you organize the state register bit values so that one bit in the state register determines whether the machine is in a state of interest. Take, for example, the following sequence of states in which it is required that the *Cn* states are identified:

State Name	State Register Bit Values Q3	Q2	Q1
A	0	0	0
B	0	0	1
C1	1	0	1
C2	1	1	1
C3	1	1	0
D	0	1	0

This choice of state register bit values allows *Q3* to be used as a flag to indicate when the machine is in any of the *Cn* states. Whenever *Q3* is high, the machine is in one of the *Cn* states. *Q3* can be assigned directly to an output pin on the device. Notice also that these bit values change by only one bit as the machine cycles through the states, as is recommended in section 3.1.5.

Combine the Use of Equations and State Diagrams 3.1.7

State diagrams can be simplified by pulling transitions that are common to every state out of the diagram and putting them into the equations section of the source file. For example, listing 3–1 shows a modified version of the decimal counter discussed in section 2.5. In every state of the counter, if the counter is in clear mode, the next state is *S0*. *S0* corresponds to the bit values [1,1,1,1]. Rather than putting the transition condition,

```
(Mode == Clear) : S0;
```

in every state, that transition is written as the equation,

```
! [Q3,Q2,Q1,Q0] := (Mode == Clear) & ! [1,1,1,1];
```

Whenever the counter is in clear mode, this equation sets *Q3–Q0* to 1, which forces the counter to state *S0*. This equation employs a clocked assignment with the symbol ":=" so that the transition occurs on a clock pulse, as is required. The equation is ORed with the equations produced by the state diagram to provide the correct function. (See "Multiple Assignments to the Same Identifier" in section 3.2 of this manual, and in section 2.8.6 of the ABEL Language Reference.)

Writing equations that work with a state diagram can save time when the source file is processed by the language processor and can save product terms.

MULTIPLE ASSIGNMENTS TO THE SAME SIGNAL 3.2

When a signal (output pin or node) name appears on the left side of more than one equation, the two equations are ORed together to produce an equation that fully describes the logic function. You can use this ORing of equations to split a design description into functional pieces, each of which describes a distinct part of the design; these individual pieces are ORed together to describe the whole design.

Listing 3–2 shows a 4–bit counter with a 2–input multiplexer described by two separate equations sections. (Equations sections are begun by the keyword EQUATIONS.) The first equations section describes the multiplexing function, and the second describes the count function. Notice, however, that both groups of equations are written for the same outputs, Q0–Q3. The multiplexing equations for Q0 are ORed with the count equations for Q0, and together they describe the total function for that output. The same operation is performed for the other outputs to completely describe the design.

This is the same 4–bit counter described in section 2.4, but in that section the count and multiplexing functions are described together in one equations section. The function of each counter is identical, and the same equations are produced regardless of the method of description chosen.

```
module count4a
title '4-bit counter with 2 input mux
based on an example by Birkner/Coli in the MMI PAL Handbook
Dan Burrier     Data I/O  Redmond WA       24 Feb 1984'

        P7022    device 'P16R4';

        Clk,OC                          pin  1,11;
        A0,A1,A2,A3,B0,B1,B2,B3         pin  2,3,4,5,6,7,8,9;
        Q3,Q2,Q1,Q0                     pin  14,15,16,17;
        CO,I1,I0,CI                     pin  12,13,18,19;

                        H,L,X,Z,C       = 1,0, .X.,.Z.,.C.;
                        InputA          = [A3,A2,A1,A0];
                        InputB          = [B3,B2,B1,B0];
                        Output          = [CO,Q3,Q2,Q1,Q0];
                        Mode            = [I1,I0];

        Hold,LoadA,LoadB,Incr           = 0,1,2,3;      " define Modes

equations       " input multiplexer

        !Q0     := (Mode==Hold ) & !Q0
                 # (Mode==LoadA) & !A0
                 # (Mode==LoadB) & !B0;

        !Q1     := (Mode==Hold ) & !Q1
                 # (Mode==LoadA) & !A1
                 # (Mode==LoadB) & !B1;

        !Q2     := (Mode==Hold ) & !Q2
                 # (Mode==LoadA) & !A2
                 # (Mode==LoadB) & !B2;

        !Q3     := (Mode==Hold ) & !Q3
                 # (Mode==LoadA) & !A3
                 # (Mode==LoadB) & !B3;

equations       " 4 bit counter

        !Q0     := (Mode==Incr ) & !CI & !Q0          "Hold if no carry
                 # (Mode==Incr ) &  CI &  Q0 ;

        !Q1     := (Mode==Incr ) & !CI & !Q1          "Hold if no carry
                 # (Mode==Incr ) & !Q0 & !Q1          "Hold if Q0=L
                 # (Mode==Incr ) &  CI &  Q0 &  Q1 ;

        !Q2     := (Mode==Incr ) & !CI & !Q2          "Hold if no carry
                 # (Mode==Incr ) & !Q0 & !Q2          "Hold if Q0=L
                 # (Mode==Incr ) & !Q1 & !Q2          "Hold if Q1=L
                 # (Mode==Incr ) &  CI &  Q0 &  Q1 &  Q2 ;

        !Q3     := (Mode==Incr ) & !CI & !Q3          "Hold if no carry
                 # (Mode==Incr ) & !Q0 & !Q3          "Hold if Q0=L
                 # (Mode==Incr ) & !Q1 & !Q3          "Hold if Q1=L
                 # (Mode==Incr ) & !Q2 & !Q3          "Hold if Q2=L
                 # (Mode==Incr ) &  CI &  Q0 &  Q1 &  Q2 &  Q3 ;

        !CO     = !CI # !Q0 # !Q1 # !Q2 # !Q3 ;
```

Listing 3–2. Multiple Equations Sections, 4–Bit Counter

Listing continued on next page.

```
@page
test_vectors  ' test Load A and B'
      ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
       [ C, L, LoadA,  ^h0  ,  ^hF  ,X  ] -> ^h0;
       [ C, L, LoadB,  ^h0  ,  ^hF  ,X  ] -> ^hF;
       [ C, L, LoadA,  ^h1  ,  ^h7  ,X  ] -> ^h1;
       [ C, L, LoadB,  ^h1  ,  ^h7  ,X  ] -> ^h7;
       [ C, L, LoadA,  ^h2  ,  ^hB  ,X  ] -> ^h2;
       [ C, L, LoadB,  ^h2  ,  ^hB  ,X  ] -> ^hB;
       [ C, L, LoadA,  ^h4  ,  ^hD  ,X  ] -> ^h4;
       [ C, L, LoadB,  ^h4  ,  ^hD  ,X  ] -> ^hD;
       [ C, L, LoadA,  ^h8  ,  ^hE  ,X  ] -> ^h8;
       [ C, L, LoadB,  ^h8  ,  ^hE  ,X  ] -> ^hE;
       [ C, L, LoadA,  ^h0  ,  ^hF  ,X  ] -> ^h0;
       [ C, L, LoadB,  ^h0  ,  ^hF  ,X  ] -> ^hF;

test_vectors  ' test increment'
      ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
       [ C, L, LoadB,   X   ,  ^h1  ,X  ] -> ^h1;
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^h2;
       [ C, L, LoadB,   X   ,  ^h3  ,X  ] -> ^h3;
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^h4;
       [ C, L, LoadA,  ^h7  ,   X   ,X  ] -> ^h7;
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^h8;
       [ C, L, LoadA,  ^hF  ,   X   ,X  ] -> ^hF;
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^h0;      "roll over
       [ C, L, LoadB,   X   ,  ^hC  ,X  ] -> ^hC;
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^hD;
       [ C, L, Hold ,   X   ,   X   ,H  ] -> ^hD;

test_vectors  ' test carry'
      ([Clk,OC, Mode, InputA, InputB,CI ] -> Output)
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^hE;
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^h1F;     "carry out
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^h0;      "roll over
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^h1;
       [ C, L, Incr ,   X   ,   X   ,L  ] -> ^h1;      "no carry in
       [ C, L, Incr ,   X   ,   X   ,H  ] -> ^h2;
       [ L, H, Hold ,   X   ,   X   ,X  ] -> [X,Z,Z,Z,Z];
end count4a
```

Listing 3–2. *(Continued)*

HAZARD–FREE LOGIC DESIGN AND "REDUCE" 3.3

Timing problems sometimes occur in logic circuits where propagation delays vary throughout the design or where one path to an output is longer or shorter than others. Many methods exist to eliminate such hazards to produce "hazard–free" logic, one of them being the introduction of redundancy into the circuit. If you use this technique, you must be aware that the reduction pass of the language processor, when run at reduction level 2, is designed to eliminate redundancy—whether it was intentional or not. Thus, if you run REDUCE (the reduction pass) at –r2 (reduction level 2) or at –r3 (reduction level 3), you will counteract any attempts at solving timing problems by the introduction of redundancy. Reduction levels 0 and 1 do not eliminate redundant terms and may be used safely with intentionally redundant logic.

Figure 3–1 shows a circuit that has a timing problem because both the complement of the signal C and the uncomplemented signal are used. Notice that the part of the circuit using !C has one additional unit of propagation delay.

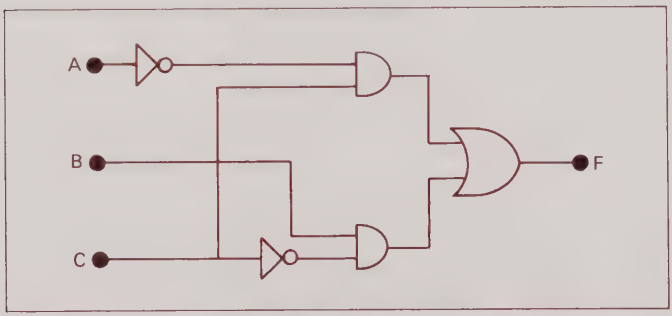


Figure 3–1. Circuit Using an Input and Its Complement

Thus the output *F* defined by the equation,

```
F = B & !C # !A & C
```

which uses both C and its complement experiences a glitch as C undergoes a transition from 1 to 0, as shown in the timing diagram in figure 3–2.

The hazard can be eliminated by the introduction of a redundant term[1], *! A & B*, so that the full equation becomes,

```
F = B & !C # !A & C # !A & B
```

This new equation does, in fact, remove the timing problem. But, if the language processor is run with level 2 or level 3 reduction in effect, the redundant term will be eliminated, resulting in the original equation with the hazard present.

Information on the language processor, the reduction pass, and the different reduction levels is presented in sections 4.1 and 4.4 of the ABEL User's Guide.

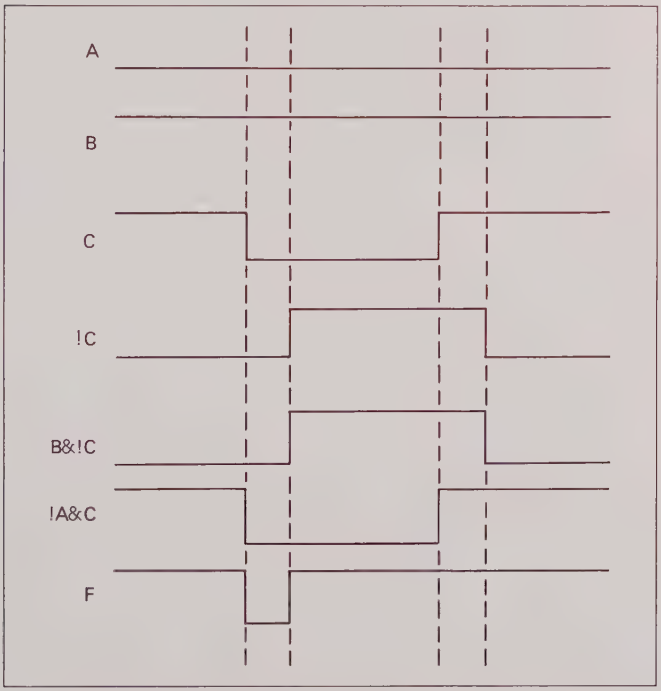


Figure 3–2. Timing Diagram for F = B & !C # !A & C

[1]Harris Programmable Logic (HPL™) Application Note 106, Hazard Free Logic Design, Steven Bennett.

PASSING ARGUMENTS FROM THE COMMAND LINE 3.4

As discussed in the ABEL™ Language Reference, dummy arguments can be used within a source file so that variable values can be passed from the command line when the ABEL™ language processor is invoked. This section describes such a use of dummy arguments to provide a variable device type to the memory address decoder presented in section 2.1.

Device types must always be chosen explicitly before an ABEL™ logic description can be converted to a programmer load file. This requirement arises because the language processor uses device-specific information to convert logic descriptions to programmer load files and to check designs for compatibility with the device. The processor detects design errors such as too many product terms for the chosen device, too many inputs or outputs and other device-dependent restrictions. The device type is specified in the declarations section of the source file with a device declaration.

The situation could exist in which many devices are suited to a particular logic design problem but in which the availability of any certain part is uncertain, or where power requirements dictate the use of one part in one system design and another in a different system—both to provide the same function. Given such a situation, it would be convenient to specify the device type when the source file is processed rather than having to edit the file before each run.

This idea is used in the example logic design, M6809D, shown in listing 3-3. M6809D allows the user to specify the device type from the language processor command line rather than in the source file itself. Thus, the same source file can be used to create many different programmer load files for different devices. M6809D also provides a default device type so that if none is specified, a P14L4 is used by default. Dummy arguments and directives are used to create the variable device type, as follows.

The variable device type is implemented in three of the first eleven lines of the source file. These three lines are discussed in detail here; the remainder of the source file is described in section 2.1.

Line 1:

```
module M6809D (dev);
```

The first line of the source file is a MODULE statement that names the module *M68O9D* and indicates that a dummy argument, *dev*, is to be used within the module. When the language processor is invoked, an argument can be passed to the module, as shown here:

```
abel m6809d -aP16L8
```

This command line invokes the language processor to process the file named *m68O9d.abl*. The argument *P16L8* will be substituted for *dev* wherever *dev* is preceded by a question mark in the source file. (A P16H8 or F82S153 can also be used.) The question mark is a required marker that must precede all dummy arguments where values are to be substituted in their place.

Line 8:

```
@ ifnb (?dev) { U09 device '?dev';
                @ message 'Using ''?dev''.' };
```

Line 8 uses the "IF NOT BLANK" directive, @ IFNB, to insert the appropriate device declaration into the source file. If the value passed for *dev* is not blank, the text enclosed by the left and right braces is inserted into the source with the value of *dev* substituted. The text within the braces is a standard device declaration followed by the @ MESSAGE directive. @ MESSAGE causes the text within single quotes to be written to the terminal. In this case, the message indicates what device is being used.

For example, if the argument *P16L8* were passed from the command line, as shown above, line 8 would cause,

```
U09 device 'P16L8';
```

to be inserted into the source for processing.

Line 10:

```
@ ifb (?dev) { U09 device 'P14L4';
               @ message 'Using "P14L4".'};
```

Line 10 functions similarly to line 8 but covers the opposite situation: If no device type is passed from the command line, the device type of P14L4 is used. This is because *?dev* would be blank, and the text enclosed by braces would be inserted into the source.

Lines 1, 8, and 10 allow the user to pass a device type to the source file when it is processed. If no device type is specified, a default value is used. Arguments can be passed to any source file in a similar manner. The dummy argument must be specified in the MODULE statement, the argument is preceded by a question mark in the module wherever substitution is desired, and the argument value is passed to the source when the language processor is invoked.

```
module m6809d (dev)
title '6809 memory decode
Jean Designer    Data I/O Corp Redmond WA   24 Feb 1984'

" The device type may be specified on the command line with
" ABEL m6809d -aP16L8

@ifnb (?dev) { U09 device '?dev';   @message 'Using "?dev".'};

@ifb  (?dev) { U09 device 'P14L4';  @message 'Using "P14L4".'};

        A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
        ROM1,IO,ROM2,DRAM       pin 14,15,16,17;

        H,L,X   = 1,0,.X.;
        Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

equations
        !DRAM   = (Address <= ^hDFFF);

        !IO     = (Address >= ^hE000) & (Address <= ^hE7FF);

        !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);

        !ROM1   = (Address >= ^hF800);

test_vectors (Address -> [ROM1,ROM2,IO,DRAM])
               ^h0000  -> [  H,  H,  H,  L ];
               ^h4000  -> [  H,  H,  H,  L ];
               ^h8000  -> [  H,  H,  H,  L ];
               ^hC000  -> [  H,  H,  H,  L ];
               ^hE000  -> [  H,  H,  L,  H ];
               ^hE800  -> [  H,  H,  H,  H ];
               ^hF000  -> [  H,  L,  H,  H ];
               ^nF800  -> [  L,  H,  H,  H ];
end m6809d
```

Listing 3-3. Device Type Passed From Command Line, Memory Address Decoder

EFFECT OF EQUATION POLARITY ON REDUCTION SPEED 3.5

When designing with ABEL, it is important to consider the impact of equation polarity on reduction speed. For example, the following is an active-high, sum-of-products equation describing an eight-into-one multiplexer function:

```
Output = A & S2 & S1 & S0
       # B & S2 & S1 & S0
       # C & S2 & !S1 & S0
       # D & S2 & !S1 & !S0
       # E & !S2 & S1 & S0
       # F & !S2 & S1 & !S0
       # G & !S2 & !S1 & S0
       # H & !S2 & !S1 & !S0;
```

This equation, as written, will fit into an active-high device, such as P16H2, with no reduction required.

However, if an active-low device is specified in the design, ABEL will have to change the polarity of the equation, performing a large number of DeMorgan expansions and reductions to produce an equation in active-low, sum-of-products form.

To avoid potentially long logic reduction times, write active low forms of equations, or use active low declarations as described in section 3.5.2 of the Language Reference manual.

SIMULATION 4

The simulation facility is a powerful feature of ABEL™ that allows you to find and correct design errors before programmable logic devices are programmed. This can mean great savings in time and expense during the logic design cycle. By simulating the operation of a part, you can make sure that a design is correct before you program the part, thus eliminating trial-and-error programming and testing of devices. Simulation is performed by the SIMULATE step of the language processor.

SIMULATE simulates the actual operation of a programmed device, as opposed to simply simulating the logic description provided in the source file. This is an important distinction. Simple simulation of the logic description, as contained in the source file, would not take the device that is to be programmed into account, and, clearly, such simulation would not truly indicate whether the programmed device would function as desired. To simulate the real operation of the device, device specific information must be used in conjunction with the programming information. This is how ABEL™ performs simulation.

SIMULATE does not evaluate the Boolean equations, truth tables, or state diagrams contained in the source file. SIMULATE uses the test vectors that you provide, the programmer load file created by previous steps of the language processor, and device characteristics to simulate the operation of the programmed device.

The information contained in this section explains some of the finer points of using the simulation facility and gives some suggestions concerning its use. Each subsection covers a separate topic. The subsections are:

4.1 Test Vectors and Simulation
4.2 Trace Levels and Breakpoints
4.3 Debugging State Machines
4.4 Multiple Test Vector Tables
4.5 Using Macros and Directives to Create Test Vectors
4.6 Don't Cares in Simulation
4.7 Preset, Reset and Reload Registers
4.8 Asynchronous Circuits

TEST VECTORS AND SIMULATION 4.1

SIMULATE simulates the operation of a programmed device, but the simulation output is only as good as the input you provide with your test vectors. With test vectors, you specify the required outputs for given input combinations. SIMULATE applies the inputs specified in the test vectors to your design and compares the required outputs with the actual outputs; if there is a difference between the required and actual outputs, an error is indicated. However, if you do not specify outputs for some combination of inputs, that combination of inputs is not applied to the design, and therefore is not tested. It is to your advantage to create complete sets of test vectors that test all functions of your logic design, and to use SIMULATE regularly as design changes are made.

TRACE LEVELS AND BREAKPOINTS 4.2

Through careful use of trace levels and breakpoints, you can easily control the amount of information that SIMULATE provides and make the analysis of a faulty design much easier. SIMULATE provides everything from simple error messages indicating that the actual outputs differ from the outputs you predicted in your test vectors to detailed information about the states of internal registers and gates of a device during simulation. It is suggested that you first simulate a design at the lowest trace level (level 0) to determine whether any errors exist. Once you find errors, use the higher trace levels (1, 2 or 3) successively to increase the amount of information provided until you have enough information to solve the problem. Examples of the output produced by the different trace levels are given in section 4.6 of the User's Guide.

With small designs, you can rerun SIMULATE with different trace levels to obtain the level of information you desire or need. With a larger or more complex design with many test vectors, however, this may result in so much information that you have difficulty finding that which is pertinent to the error. This is when breakpoints become helpful. Assume that you have run a simulation and detected an error in the twentieth test vector out of fifty vectors. You want to see more information to determine the cause of the error. If you simply rerun SIMULATE with a higher trace level, more detailed information will be collected for every one of the fifty test vectors, when in fact you only need detailed information for the vector 20. If you run SIMULATE with breakpoints set before and after test vector 20, the higher level of information will be provided only for that vector, thus reducing significantly the amount of information collected.

Detailed information on the different trace levels and on using breakpoints is provided in section 4.6 of the ABEL™ User's Guide. In summary:

- Simulate designs at trace level 0 to determine the existence of errors.
- Once an error is found, increase the trace level until you have enough information to correct the error.
- Use breakpoints to limit simulation data collection to the vectors of concern.

DEBUGGING STATE MACHINES 4.3

State machines can be difficult to debug once an error occurs because each state is dependent on previous states and points to next states. A simple error in the description of one state transition can cause an incorrect output that cascades through the state machine causing so many errors during simulation that the original error is difficult to isolate. Therefore, when you debug state machines of any size, you should periodically force (with your test vectors) the machine to a known state and then let the state transitions take place. In this manner, you limit the cascading of errors to a smaller number of states and make it much easier to find initial errors.

It is also helpful with large state machines to start with a small set of test vectors that tests only part of the state machine's function. When operation of that function has been verified, add a set of test vectors that test another function, then add another, and so on, until you have tested the full function of the state machine. Combining this technique of gradual simulation with the forcing vectors discussed above greatly simplifies the testing of large state machines.

In summary, to test and debug larger state machines:

- Write small sets of test vectors that test individual functions of the state machine, and gradually add them to the simulation.
- Add test vectors that periodically force the machine to known states to eliminate cascading of errors.

MULTIPLE TEST VECTOR TABLES 4.4

More than one set of test vectors can be used to simulate the function of a device. This may be useful in the representation of the test in the source file. Take for example, the source file presented in listing 4–1 that describes AND and NAND gates implemented on the same device as well as the test vectors used to simulate the operation of that device. The test vectors are described in two separate sections. The first test vectors section lists the test vectors that simulate the operation of the AND portion of the design, and the second section tests the NAND function. With the test vectors written as they are, in two separate sections, the correspondence between test vectors and the function being tested is readily apparent in the source file.

In a similar manner, anytime you have two or more distinct functions being performed by the same device, you may want to describe the vectors in separate sections for each function. You are not, however, required to do so.

```
module simple
title 'Simple ABEL example
Jane Engineer   Data I/O Corp   24 Feb 1984'

                U7      device  'P14H4';
        A1,A2,A3        pin 1,2,3;
        N1,N2,N3        pin 4,5,6;
        AND,NAND        pin 14,15;

equations
         AND    = A1 & A2 & A3;

        !NAND   = N1 & N2 & N3;

test_vectors  ( [A1,A2,A3] -> AND )     " Test And Gate
                [ 0, 0, 0] ->  0;
                [ 1, 0, 0] ->  0;
                [ 0, 1, 0] ->  0;
                [ 0, 0, 1] ->  0;
                [ 1, 1, 1] ->  1;

test_vectors  ( [N1,N2,N3] -> NAND )    " Test Nand Gate
                [ 0, 0, 0] ->  1;
                [ 1, 0, 0] ->  1;
                [ 0, 1, 0] ->  1;
                [ 0, 0, 1] ->  1;
                [ 1, 1, 1] ->  0;

end simple
```

Listing 4–1. Source File With Multiple Test Vectors Sections

USING MACROS AND DIRECTIVES TO CREATE TEST VECTORS 4.5

Macros and directives can be used to write test vectors in a concise form that is often helpful in large designs that require many test vectors to fully test the device operation. In this section, two examples of using macros and directives to create test vectors are shown for a simple design. The advantages gained through writing test vectors in this way are even greater for more complicated designs.

Listing 4-2 shows a modified version of the memory address decoder presented in section 2.1. In this version, a macro and the @ IRP directive are used to create the test vectors. The macro *between* is defined after the constant and set declarations and before the equations section. The *between* macro uses the @ IF directive to produce the character "L" if an address is in a given range and the character "H" if the address is not in that range.

```
module m6809b
title '6809 memory decode
Jean Designer     Data I/O Corp Redmond WA   24 Feb 1984'

                          U09      device   'P14L4';
        A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
        ROM1,IO,ROM2,DRAM       pin 14,15,16,17;

        H,L,X    = 1,0,.X.;
        Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

" This macro will return an 'L' if the address (a) is between the
" two limits (b,c), otherwise it returns an 'H'.
between macro (a,b,c)
            {@if ((?a>=?b)&(?a<=?c)){L}@if!((?a>=?b)&(?a<=?c)){H}};

equations
        !DRAM    = (Address <= ^hDFFF);

        !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);

        !ROM2    = (Address >= ^hF000) & (Address <= ^hF7FF);

        !ROM1    = (Address >= ^hF800);

test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
        @IRP addrs (^h0000,^h8000,^hE100,^hE800,^hF100,^hFC00) {

                ?addrs  -> [ between (?addrs,^h0000,^hDFFF),     " DRAM
                             between (?addrs,^hE000,^hE7FF),     " IO
                             between (?addrs,^hF000,^hF7FF),     " ROM2
                             between (?addrs,^hF800,^hFFFF)];    " ROM1}
end m6809b
```

Listing 4-2. Test Vectors Described With a Macro and @IF and @IRP Directives: Memory Address Decoder

The macro has three dummy arguments, *a, b* and *c*, whose values are supplied when the macro is invoked. Argument *a* represents the address that either falls or does not fall in the range between arguments *b* and *c*. The body of the macro is specified in the block defined by the outermost left and right braces. Within the block, the @ IF directive is used to check the value supplied for *a* against the range values supplied for *b* and *c*. The @ IF directive itself contains blocks that contain the "L" or "H" character. If the condition in parentheses following the @ IF directive is true, the block following the condition is inserted into the text. Thus, there are two @ IF directives: the first produces an "L" if the address is in the range; the second produces an "H" if the address is out of the range.

Now that the *between* macro has been defined, it can be used in the test vectors section to help build the test vectors. The @ IRP directive invokes *between* to insert the "L" or "H" needed to properly define the vectors. @ IRP causes the block following it (that text enclosed by braces), to be repeated once for each value contained in the parentheses in the @ IRP statement. Each time the block is repeated, one of those values is successively substituted for the dummy argument *addrs* declared in the @ IRP statement. This means that the block will be repeated six times, and *addrs* will take on six different values each time *between* is invoked in the block. *addrs* is substituted for *a* in the macro and the values for *b* and *c* are given explicitly. The resulting source is shown in the expanded listing produced by the PARSE step of the language processor. This listing, shown in listing 4–3, shows how the directive and macro were expanded to produce the test vectors. The listing was produced by PARSE with the –E parameter.

```
ABEL Version 1.00 Copyright (c) 1983,1984 Data I/O Corp.

0001e:module m6809b
0002e:title '6809 memory decode
0003e:Jean Designer     Data I/O Corp Redmond WA   24 Feb 1984'
0004e:
0005e:                                  J09       device  'P14L4';
0006e:          A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
                ROM1,IO,ROM2,DRAM       pin 14,15,16,17;
                ...
0009e:          H,L,X    =1,0,.X.,
0010e:          Address =[A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];
0011e:
0012e:" This macro will return an 'L' if the address (a) is between the
0013e:" two limits (b,c), otherwise it returns an 'H'.
0014e:between macro (a,b,c)
0015e:          {@if ((?a)>=?b)&(?a<=?c)){L}@if!((?a)>=?b)&(?a<=?c)){H}};
0016e
0017e:equations
0018e:          !DRAM    = (Address <= ^hDFFF);
0019e:
0020e:          !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);
0021e:
0022e:          !ROM2    = (Address >= ^hF000) & (Address <= ^hF7FF);
                ...
                !ROM1    = (Address >= ^hF800).
                ...
0035e:test vectors (Address -> [DRAM, IO, ROM2, ROM1])
                ...
                ...
0029e:                  ^h0000  -> [ L,     " DRAM
0030e:                               H,     " IO
0031e:                               H,     " ROM2
0032e:                               H];    " ROM1
0033e:
0034e:                  ^h8000  -> [ L,     " DRAM
0035e:                               H,     " IO
0036e:                               H,     " ROM2
0037e:                               H];    " ROM1
0038e:
0039e:                  ^hE100  -> [ H,     " DRAM
0040e:                               L,     " IO
0041e:                               H,     " ROM2
0042e:                               H];    " ROM1
0043e:
0044e:                  ^hE800  -> [ H,     " DRAM
0045e:                               H,     " IO
0046e:                               H,     " ROM2
0047e:                               H];    " ROM1
0048e:
0049e:                  ^hF100  -> [ H,     " DRAM
0050e:                               H,     " IO
0051e:                               L,     " ROM2
0052e:                               H];    " ROM1
0053e:
0054e:                  ^hFC00  -> [ H,     " DRAM
0055e:                               H,     " IO
0056e:                               H,     " ROM2
0057e:                               L];    " ROM1
0058e:end m6809b
0059e:
```

Listing 4–3. PARSE Output Showing Expanded Output for a Macro and @IF and @IRP Directives: Memory Address Decoder

In the second source file, shown in listing 4–4, the @ CONST and @ REPEAT directives are used in conjunction with the *between* macro to create the test vectors for the same design. The definition of *between* is identical to that used in the previous example. *@ REPEAT 6* causes the block containing the vectors to be repeated 6 times. Within that block, @ CONST is used to increment the value of *addrs* before the next repetition of the block. Listing 4–5 shows the expanded output produced by the PARSE step of the language processor invoked with the –E parameter. Notice that in the test vectors shown in the listing, the constant *addrs* appears instead of the actual numeric value that the constant represents. The correct value is retained internally, and these test vectors are functionally the same as those shown in listing 4–3.

```
module m6809c
title '6809 memory decode
Jean Designer     Data I/O Corp Redmond WA   24 Feb 1984'

                        U09     device  'P14L4';
        A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
        ROM1,IO,ROM2,DRAM       pin 14,15,16,17;

        H,L,X   = 1,0,.X.;
        Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

" This macro will return an 'L' if the address (a) is between the
" two limits (b,c), otherwise it returns an 'H'.
between macro (a,b,c)
            {@if ((?a>=?b)&(?a<=?c)){L}@if!((?a>=?b)&(?a<=?c)){H}};

equations
        !DRAM   = (Address <= ^hDFFF);

        !IO     = (Address >= ^hE000) & (Address <= ^hE7FF);

        !ROM2   = (Address >= ^hF000) & (Address <= ^hF7FF);

        !ROM1   = (Address >= ^hF800);

test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
        @CONST  addrs = ^hD000;
        @REPEAT 6 {
                addrs   -> [ between (addrs,^h0000,^hDFFF),     " DRAM
                             between (addrs,^hE000,^hE7FF),     " IO
                             between (addrs,^hF000,^hF7FF),     " ROM2
                             between (addrs,^hF800,^hFFFF)];    " ROM1
        @CONST  addrs = addrs + ^h800;}
end m6809c
```

Listing 4–4. Test Vectors Described With a Macro, @ CONST and @ REPEAT Directives: Memory Address Decoder

```
ABEL Version 1.00 Copyright (c) 1983,1984 Data I/O Corp.

0001e|module m6809c
0002e|title '6809 memory decode
0003e|Jean Designer     Data I/O Corp Redmond WA   24 Feb 1984'
0004e|
0005e|                              U09      device  'P14L4';
0006e|         A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
0007e|         ROM1,IO,ROM2,DRAM       pin 14,15,16,17;
0008e|
0009e|         H,L,X  =1,0,.X.;
0010e|         Address=[A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];
0011e|
0012e|" This macro will return an 'L' if the address (a) is between the
0013e|" two limits (b,c), otherwise it returns an 'H'.
0014e|between macro (a,b,c)
0015e|           [@if ((?a>=?b)&(?a<=?c)){L}@if!((?a>=?b)&(?a<=?c)){H}};
0016e|
0017e|equations
0018e|         !DRAM    = (Address <= ^hDFFF);
0019e|
0020e|         !IO      = (Address >= ^hE000) & (Address <= ^hE7FF);
0021e|
0022e|         !ROM2    = (Address >= ^hF000) & (Address <= ^hF7FF);
0023e|
0024e|         !ROM1    = (Address >= ^hF800);
0025e|
0026e|test_vectors (Address -> [DRAM, IO, ROM2, ROM1])
0027e|
0028e|
0029e|                  addrs    -> [ L,      " DRAM
0030e|                                H,      " IO
0031e|                                H,      " ROM2
0032e|                                H];     " ROM1
0033e|
0034e|                  addrs    -> [ L,      " DRAM
0035e|                                H,      " IO
0036e|                                H,      " ROM2
0037e|                                H];     " ROM1
0038e|
0039e|                  addrs    -> [ H,      " DRAM
0040e|                                L,      " IO
0041e|                                H,      " ROM2
0042e|                                H];     " ROM1
0043e|
0044e|                  addrs    -> [ H,      " DRAM
0045e|                                H,      " IO
0046e|                                H,      " ROM2
0047e|                                H];     " ROM1
0048e|
0049e|                  addrs    -> [ H,      " DRAM
0050e|                                H,      " IO
0051e|                                L,      " ROM2
0052e|                                H];     " ROM1
0053e|
0054e|                  addrs    -> [ H,      " DRAM
0055e|                                H,      " IO
0056e|                                H,      " ROM2
0057e|                                L];     " ROM1
0058e|
0059e|end m6809c
0060e|
```

Listing 4-5. PARSE Output Showing Expanded Output for a Macro and @ CONST and @ REPEAT Directives: Memory Address Decoder

DON'T CARES IN SIMULATION 4.6

In ABEL you can use the special constant .X. in a test vector to denote a don't care input or output. The .X. tells the ABEL simulator to choose a value for the input designated by the .X. in the test vector(s). The default value used in the simulator for the don't care inputs is zero; however, the don't care flag (-Xn) can be used in the SIMULATE command line to specify zero or one (0 or 1) for the don't care value. (Refer also to section 4.6 of the User's Guide.)

Input pins that are not specified in the test vectors are given the default don't care value zero (0) by the simulator unless the don't care flag is set to -X1. In this case, all unspecified pins will be assigned a value of 1 in the test vectors.

If you experience trouble with devices not working in a circuit or programmer/tester, it may be helpful to recheck the don't care assumptions. There may be a combination of 1's and 0's in a test vector that needs to be checked by the ABEL simulator. (The Data I/O LogicPak does not test the device in the same manner as the ABEL simulator. If the LogicPak displays ERROR 75 you may determine the failed vector and pins by using the programmer/LogicPak in the terminal mode as described in the LogicPak manual.)

Also, the simulator checks the design with a single level for the don't care inputs, while the target circuit may place other levels on the input during actual operation. For complete simulation, it is recommended that you run the SIMULATE operation with the don't cares set to 0 (flag -X0), and then again with them set to 1 (flag -X1).

Output pins that are not specified in the test vectors are disregarded by the simulator and no error will be indicated due to conflict between a specified value and the value determined by the simulator. The .X. constant at an output pin tells the simulator not to compare the outputs (the output produced by the design and the output specfied in the test vector) but still allow them to be displayed. Listing 4-6 shows how the .X. value can be assigned to the outputs prior to the SIMULATE step of the language processor.

Using trace level 1, 2, or 3, you can observe the actual output values determined by the simulator. In listing 4-7, the X entries (in the test vectors) for pins 14 and 15 allow the simulator to display an H or L to indicate the output value for the specified inputs. (All unused inputs are set to the default condition, 0.) The N entries indicate outputs that are not to be tested.

```
module findout  flag '-tl'

title 'The ABEL simulator will find the output levels
Ngoc Nicholas    FutureNet - Data I/O    8 Nov 1985'

        F1      device 'P16L8';

        A,B,Y1,Y2       pin 1,2,14,15;

        X = .X.;

equations

        !Y1 = A # B;
        !Y2 = A & B;

test_vectors
        ([A,B]  -> [Y1,Y2])
        [0,0]  -> [ X, X],
        [0,1]  -> [ X, X],
        [1,0]  -> [ X, X];
        [1,1]  -> [ X, X];
end
```

Listing 4-6. Assignment of Don't Care Value (.x.) to Design Outputs

```
ABEL Version 2.00  Data I/O Corp.

The ABEL simulator will find the output levels
Ngoc Nicholas    FutureNet - Data I/O    8 Nov 1985


Simulate device F1, type 'P16L8'

Vector 1
Output                                -> |..........ZZHHZZZZ.|
V 0001   [000000000-000000000-] -> |..........NNXXNNNN.|
                                  PIN 14 ——▶ ◀—— PIN 15
Vector 2
Output                                -> |..........ZZLLZZZZ.|
V 0002   [010000000-000000000-] -> |..........NNXXNNNN.|

Vector 3
Output                                -> |..........ZZLLZZZZ.|
V 0003   [100000000-000000000-] -> |..........NNXXNNNN.|

Vector 4
Output                                -> |..........ZZLHZZZZ.|
V 0004   [110000000-000000000-] -> |..........NNXXNNNN.|
```

Listing 4-7. SIMULATE Results with Outputs Specified as Don't Care

PRESET AND PRELOAD REGISTERS 4.7

Preset, reset, and preload are terms used to define a specific action and resultant output of one or more registers contained in a programmable logic device. Preset forces all register outputs to one, reset forces all outputs to zero, and preload forces all registers to specified states. Synchronous preset, reset, and preload functions require a clock input. Asynchronous functions require no clock input.

Several programmable logic devices contain registers that can be preset, reset, or preloaded. To verify the operation of these devices, appropriate test vectors must be written and placed in the source file. These test vectors allow the SIMULATE step of the language processor to verify operation of the design by performing the required operations of these registers.

Note that it is not possible to preset and preload at the same time. Preset and preload must not contend during preload with other inputs, preset, or register functions.

Special Preset Considerations 4.7.1

Certain programmable logic devices, such as 82S105 and 82S167, do not respond to the first clock pulse following a preset condition (invoked by power-on or the preset input). These devices allow normal clocking only after a high-to-low transition of the clock input following the preset condition. This means that simulation for these devices requires an additional test vector following the preset condition just to provide the high-to-low transition of the clock input, thereby allowing normal clocking to take place without the loss of a clock pulse.

To illustrate the preset considerations for these devices, a four-state counter with clock and preset inputs is presented in listing 4-8, along with the test vectors required to properly verify the design. This counter is targeted for a circuit that provides a power-on preset condition; so the test vectors must verify operation of the counter after power-on preset as well as after the preset input has been active.

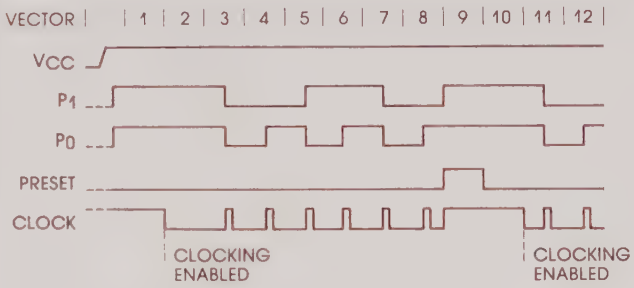
Figure 4-1 is a timing diagram that shows the action of the test vectors in listing 4-8. As indicated in figure 4-1, the preset input overrides the clock input and when held high, inhibits clocking of the counter. Assuming that the device is powered-up in the preset condition, the first test vector pulls the clock input high while the second vector pulls it low to provide the high-to-low transition on the clock line, required for normal clocking.

The next six test vectors provide clock inputs to increment the counter through all states and back to state one. As indicated in figure 4-1, the 9th test vector invokes the preset function while the 10th test vector pulls the preset input low and maintains the clock input high. The 10th test vector allows the preset line to go low before the high-to-low transition of the clock. The preset line must go low before the clock so that the high-to-low clock transition can enable the clock pulse of the 11th test vector. The high-to-low transition that follows the 10th test vector resumes normal clocking of the device.

If the 2nd and 10th test vectors are not included in the source file, the clock pulse of the 3rd and 11th vectors would be lost. That is, the high-to-low transition of the clock pulses in these vectors cause the resumption of normal clocking, but do not increment the counter as required by the design.

```
module _preset flag '-KY' "leave unused OR terms connected
title '2-bit counter to demonstrate power on preset
Michael Holley    FutureNet - Data I/O  31 Oct 1985'

        preset  device 'F82S167';

        Clk,Hold        pin  1,2;
        PR              pin  16;        "Preset/Enable
        P1,P0           pin  15,14;

        Ck      = .C.;

equations

        enable P0        = 1;   "always enabled, pin 16 is preset

        [P1.R,P0  ]     := !P1 & !P0 & !Hold;   " state 0
        [P1  ,P0.R]     := !P1 &  P0 & !Hold;   " state 1
        [P1  ,P0  ]     :=  P1 & !P0 & !Hold;   " state 2
        [P1.R,P0.R]     :=  P1 &  P0 & !Hold;   " state 3

test_vectors
       ([Clk,PR,Hold] -> [P1,P0])
        [ 1 , 0,  0 ] ->    3; " Provides a High-to-Low on clock
        [ 0 , 0,  0 ] ->    3; " to enable clocking
        [ Ck, 0,  0 ] ->    0;
        [ Ck, 0,  0 ] ->    1;
        [ Ck, 0,  1 ] ->    2;
        [ Ck, 0,  0 ] ->    3;
        [ Ck, 0,  0 ] ->    0; " Roll over
        [ Ck, 0,  0 ] ->    1;
        [ 1 , 1,  0 ] ->    3; " Preset high
        [ 1 , 0,  0 ] ->    3; " Preset low
        [ Ck, 0,  0 ] ->    0;
        [ Ck, 0,  0 ] ->    1;

" Notes on preset from the Signetics Data Sheet
"
" The PR input provides an asynchronous preset to logic '1' of all
" State and Output Register bits. Preset overrides the Clock, and
" when held High, clocking is inhibited and all outputs are High.
" Normal clocking resumes with the first clock pulse following
" a High-to-Low clock transition after PR goes Low.
"
" The power on preset also inhibites clocking until a High-to-Low
" clock transition. This is provided by the first 2 test vectors.

end
```

Listing 4-8. Test Vectors for Special Preset Functions



Preset overrides clock, and when held high, clocking is inhibited and the registers are high. Normal clocking resumes with the first clock pulse following a high-to-low clock transition after preset goes low.

Figure 4-1. Timing Diagram Showing Test Vector Action

TTL Preload 4.7.2

Certain programmable logic devices, such as the 82S159 FPLS (field programmable logic sequencer) allow internal registers to be forced (preloaded) to a known state by means of the TTL preload function. Figure 4-2 shows a typical FPLS layout. To preload the output register in such a device, four conditions must be present:

1. The output is placed in the high-impedence state.
2. The desired register state is placed on the output pin.
3. The load control term is activated.
4. A clock pulse is applied to the clock input.

The fifth test vector in listing 4-9 shows how each of the above conditions are met. In the fifth test vector:

- the OE (Ena) pin is held at 1
- the output (FO) is pulled low by inserting a 0 in the input side of the test vector
- a 1 is applied to the LOAD input which activates the load control term (LA)
- a clock pulse is provided by the C (.C.) applied to the clock input.

The sixth test vector tests to make sure the register was loaded with the 0 applied to the output by the fifth test vector. The sixth test vector enables the output (Ena at 0) and allows the output to be tested (FO = X on the input side and 0 on the output side of the vector) while holding the clock input at 0.

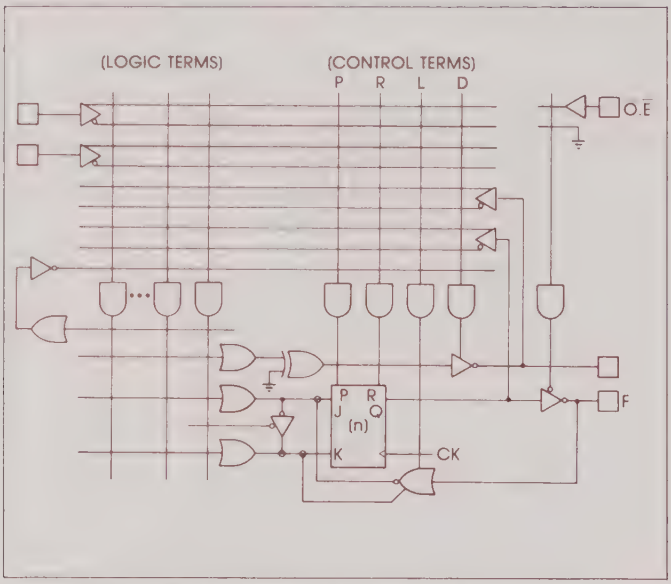


Figure 4-2. Internal Register of the 82S159

```
module TTLload
title 'TTL load example
Brenda French      FutureNet - Data I/O   21 Nov 1985'

        TTL59   device  'F82S159';

        C,L,H,X,Z       = .C.,0,1,.X.,.Z.;

        Clk,J_IN,K_IN,LOAD,Ena,FO       pin 1,2,3,4,11,12;
        LA,MO,KO                        node 27,29,37;

equations

         enable  FO  = !Ena;     " Enable controlled by pin 11
                 MO  = 0;        " fuse open - fixed at JK

                !FO := J_IN;
                 KO  = K_IN;
                 LA  = LOAD;

test_vectors
       ([Clk,Ena,J_IN,K_IN,LOAD,FO] -> FO)
        [ C , L ,  1 ,  0 ,  0 , X] ->  0 ; "Set
        [ C , L ,  0 ,  1 ,  0 , X] ->  1 ; "Reset
        [ C , L ,  1 ,  1 ,  0 , X] ->  0 ; "Toggle
        [ C , L ,  1 ,  1 ,  0 , X] ->  1 ; "Toggle
        [ C , H ,  0 ,  0 ,  1 , 0] ->  X ; "Load
        [ 0 , L ,  0 ,  0 ,  0 , X] ->  0 ; "Test
        [ C , L ,  1 ,  1 ,  0 , X] ->  1 ; "Toggle
end
```

Listing 4-9. Invoking the TTL Preload Function

Supervoltage Preload 4.7.3

Supervoltage preload allows the setting of registers, within certain devices such as the P16R4, to the logic levels placed on their registered outputs. Supervoltage preload is accomplished by means of the P test condition (.P. special constant) that is used to "jam load" registers within the logic device to the desired state. When the P test condition is applied to the clock pin, the logic level applied to the register output is loaded into the register. Devices with separate banks of registers require that the P test condition be applied to each clock pin. Also during preload, certain device pins, such as the output enable pin, may have to be in a defined state.

To verify the preload operation, use a separate test vector to test the outputs. This vector must follow the vectors that perform the preload operation.

Supervoltage preload can be used to test state machine designs that could assume one or more illegal states, or designs that contain branch conditions. An illegal state for a state machine is a state that the design does not allow, but the device is capable of assuming under certain conditions (such as power-up or noise).

The decade counter shown in listing 2-5 is defined as having ten states (states S0 through S9), but since four outputs are used, the device is capable of assuming any of six additional states (S10 through S15).

The design of this decade counter is such that whenever an illegal state is achieved, the next clock pulse returns the counter to the S0 state. During simulation it is necessary to not only test the counter for normal up/down/clear operation (performed by the test vectors) but also to insure that it will clock to state S0 from any illegal state.

To test that the counter will clock to state S0 from any illegal state, it is necessary to do two things:

1. Define all illegal states to be tested.

2. Create test vectors that verify the return to state 0 from any illegal state.

To define the illegal states for the decade counter in listing 2-5, it is necessary to expand the "Counter States" portion of the source file to define the illegal states, S10 through S15. This is accomplished by adding the entries S10 = ^b0101; S11 = ^b0100; etc., as shown in listing 4-10.

The additional test vectors will be as shown in listing 4-11. This group of test vectors preloads the device to each possible illegal state and then verifies that the device clock to state S0. The test vectors preload the counter by means of the .P. special constant applied to the clock pin and a logic high applied to the output enable (OE) pin. The test vector that follows the preload test vector verifies the result of the preload operation, while the following test vector verifies the clocking of the counter from the illegal state to state S0.

An example of a state machine that contains branch conditions is given in section 6.1 of the Applications Guide (blackjack machine). Here, it would be necessary to repeat the test vectors down to the Test22 state, for each branch of the Test22 state if it were not possible to preload the state machine to each branch condition. The test vectors in listing 4-12 show how this state machine can be preloaded to test these three branches of the design.

```
module cntl0p  flag '-r3'
title 'decimal Up/Down counter with synchronous Clear
Note: preload the data on pins into the registers
Michael Holley     FutureNet - Data I/O Corp    8 Nov 1985'

        U10             device 'P16R4';

        Clk,Clr,Dir,OE  pin 1,2,3,11;
        Q3,Q2,Q1,Q0     pin 14,15,16,17;

        Ck,X,Z,P        = .C. , .X., .Z., .P.;

" Counter States
        S0 = ^b1111;    S4 = ^b1011;    S8 = ^b0111;    S12= ^b0011;
        S1 = ^b1110;    S5 = ^b1010;    S9 = ^b0110;    S13= ^b0010;
        S2 = ^b1101;    S6 = ^b1001;    S10= ^b0101;    S14= ^b0001;
        S3 = ^b1100;    S7 = ^b1000;    S11= ^b0100;    S15= ^b0000;
```

Listing 4-10. The Illegal States Defined

```
module cnt10p  flag '-r3'
title 'decimal Up/Down counter with synchronous Clear
Note: preload the data on pins into the registers
Michael Holley     FutureNet - Data I/O Corp   8 Nov 1985'

        U10             device 'P16R4';

        Clk,Clr,Dir,OE  pin 1,2,3,11;
        Q3,Q2,Q1,Q0     pin 14,15,16,17;

        Ck,X,Z,P        = .C. , .X., .Z., .P.;

" Counter States
        S0 = ^b1111;    S4 = ^b1011;    S8 = ^b0111;    S12= ^b0011;
        S1 = ^b1110;    S5 = ^b1010;    S9 = ^b0110;    S13= ^b0010;
        S2 = ^b1101;    S6 = ^b1001;    S10= ^b0101;    S14= ^b0001;
        S3 = ^b1100;    S7 = ^b1000;    S11= ^b0100;    S15= ^b0000;

" Counter modes
        Mode    = [Clr,Dir];
        Up      = [ 1 , 1 ];
        Down    = [ 1 , 0 ];
        Clear   = [ 0 , X ];

test_vectors 'preload to illegal states'
      ( [Clk ,OE, Mode,[Q3,Q2,Q1,Q0]] -> [Q3,Q2,Q1,Q0])
        [ P   , 1, Up   ,      S10     ] ->      X ;
        [ 0   , 0, Up   ,        X     ] ->    S10;
        [ Ck  , 0, Up   ,        X     ] ->     S0 ;
        [ P   , 1, Up   ,      S11     ] ->      X ;
        [ 0   , 0, Up   ,        X     ] ->    S11;
        [ Ck  , 0, Up   ,        X     ] ->     S0 ;
        [ P   , 1, Up   ,      S12     ] ->      X ;
        [ 0   , 0, Up   ,        X     ] ->    S12;
        [ Ck  , 0, Up   ,        X     ] ->     S0 ;
        [ P   , 1, Up   ,      S13     ] ->      X ;
        [ 0   , 0, Up   ,        X     ] ->    S13;
        [ Ck  , 0, Up   ,        X     ] ->     S0 ;
        [ P   , 1, Up   ,      S14     ] ->      X ;
        [ 0   , 0, Up   ,        X     ] ->    S14;
        [ Ck  , 0, Up   ,        X     ] ->     S0 ;
        [ P   , 1, Up   ,      S15     ] ->      X ;
        [ 0   , 0, Up   ,        X     ] ->    S15;
        [ Ck  , 0, Up   ,        X     ] ->     S0 ;

@page
```

Listing 4-11. Test Vectors for Illegal States

```
test_vectors 'Test 3 way branch at Test22'
([Ena,Clk,LT22,Ace,Qstate] -> [Ace,Qstate    ])
  [ 1 ,.P.,  X , 1 ,Test22] -> [ X,     X     ];
  [ 0 , 0 ,  1 , X ,    X ] -> [ H, Test22    ]; "Verify preload
  [ 0 , C ,  1 , X ,    X ] -> [ H, ShowStand ];

  [ 1 ,.P.,  X , 0 ,Test22] -> [ X,     X     ];
  [ 0 , 0 ,  0 , X ,    X ] -> [ L, Test22    ];
  [ 0 , C ,  0 , X ,    X ] -> [ L, ShowBust  ];

  [ 1 ,.P.,  X , 1 ,Test22] -> [ X,     X     ];
  [ 0 , C ,  0 , X ,    X ] -> [ H, Sub10     ];
```

Listing 4-12. Using Test Vectors to Preload a State Machine

Preset/Reset Controlled by Product Term 4.7.4

In programmable logic devices such as the P22V10, preset and reset functions can be controlled by product terms placed in the "equations" portion of the ABEL source file. An example of controlling the preset and reset functions is given in listing 4-13. In this listing, two nodes within the P22V10 are declared as the reset and preset functions. (Nodes are internal signal lines defined by Data I/O to aid in the programming of parts with internal signals. Refer to section 5.4 for additional details on nodes.) The declared nodes are nodes 25 and 26. The device nodes appendix located at the back of the User's Guide lists nodes 25 and 26 of the P22V10 as useable for the reset and preset functions.

In the P22V10, the reset is asynchronous while preset is synchronous (to the clock input). The test vectors in listing 4-13 verify the reset and preset functions. The first three vectors verify the loading of input data. (Note that the third vector pulls the T input high; this is in preparation for T to be pulled low later in the simulation.) The fourth vector verifies operation of the clock input by changing the input data without providing a clock input. The sixth vector verifies that the R input without a low T input will not provide the asynchronous reset.

The seventh vector verifies operation of the asynchronous reset by pulling the T input low. The next vector verifies that a high S input and low T input do not preset the device without the clock input. The final vector verifies the synchronous preset by providing the clock pulse and testing the output for both output pins at logic high (decimal 3).

```
module _reset22
title 'Demonstrates Asynchronous Reset and Synchronous Preset
Dave Pellerin   FutureNet - Data I/O Corp   8 Nov 1985'

        reset22 device  'P22V10';

        Clk,I1,I2,R,S,T Pin 1,2,3,4,5,6;
        Q1,Q2           Pin 14,15;
        reset,preset    Node 25,26;

        Q2,Q1           istype 'pos';
        Ck,Z,H,L        = .C., .Z., 1, 0;
        Input           = [I2,I1];
        Output          = [Q2,Q1];
equations

        Output := Input;          "Registered buffer

        reset    = R & !T;

        preset   = S & !T;

test_vectors
       ([Clk,Input,R,S,T] -> Output)
        [ Ck,  0  ,0,0,0] ->   0;
        [ Ck,  1  ,0,0,0] ->   1;
        [ Ck,  2  ,0,0,1] ->   2;
        [ 0 ,  3  ,0,0,1] ->   2;        "Hold
        [ Ck,  3  ,0,0,1] ->   3;

        [ 0 ,  3  ,1,0,1] ->   3;        "Reset = R & !T
        [ 0 ,  3  ,1,0,0] ->   0;        "Async Reset

        [ 0 ,  0  ,0,1,0] ->   0;        "Preset requires clock
        [ Ck,  0  ,0,1,0] ->   3;        "Sync Preset

end
```

Listing 4-13. Controlling Reset/Preset by Product Term

ASYNCHRONOUS CIRCUITS 4.8

In some asynchronous circuits, the ABEL simulator may not maintain the output state and will report an error. In these cases, the device outputs should be specified on both sides of the test vector. Figure 4-3 shows a common cross-coupled flip-flop, an asynchronous circuit that can cause erroneous error reporting by the simulator.

The problem with such an asynchronous circuit is that it can power-up in an undetermined state. That is, either output can be high or low, depending on the electrical characteristics the device. In order for the simulator to verify the functionality of the circuit, it must be made to assume some beginning state for the outputs. Test vectors are used to define the output states so that the simulator can test the design.

Listing 4-14 shows two sets of test vectors that could be used to test the design of a cross-coupled flip-flop such as that shown in figure 4-3. The first set of test vectors defines the inputs only, and tests the outputs for the desired results. Because of the circuit's asynchronous nature, the simulator may not be able to maintain the output state and may report an error.

The second set of test vectors specifies the outputs on both sides of each vector. If the outputs are not defined in this manner, the simulator may report an error due to the conflict between outputs tied back to the inputs. The second set of test vectors defines the outputs for the simulator and prevents any error indications due to the asynchronous nature of the circuit.

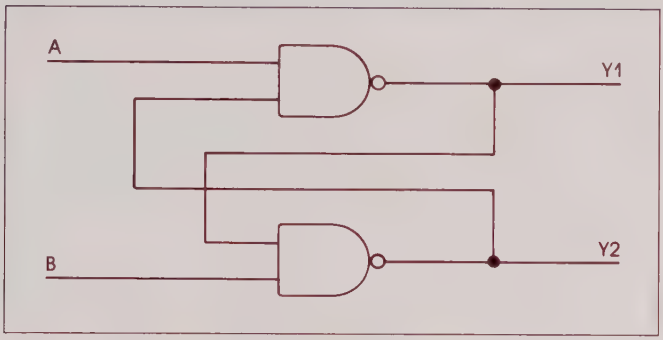


Figure 4-3. A Cross-Coupled Flip-Flop

```
module _flipflop
title 'Simulation of an asynchronous flip flop'

        flipflop        device 'P16L8';

        A,B,Y1,Y2       pin 1,2,13,14;

equations

        Y1      = !(A & Y2);    "Cross coupled flip flop
        Y2      = !(B & Y1);

test_vectors
        ([A,B] -> [Y1,Y2])
         [1,0] -> [ 0, 1];
         [1,1] -> [ 0, 1];
         [0,1] -> [ 1, 0];
         [1,1] -> [ 1, 0];

test_vectors
        ([A,B,Y1,Y2] -> [Y1,Y2])
         [1,0, 0, 1] -> [ 0, 1];
         [1,1, 0, 1] -> [ 0, 1];
         [0,1, 1, 0] -> [ 1, 0];
         [1,1, 1, 0] -> [ 1, 0];
end
```

Listing 4-14. Using the Input Side of the Test Vectors to Define Outputs

USING ADVANCED DEVICE FEATURES 5

The following paragraphs provide information that will help you utilize the features of some of the more advanced programmable logic devices. Subjects dealt with in this section include devices with enable-type outputs, macro cells, J-K and D-type outputs, and internal nodes.

OUTPUT ENABLES 5.1

There are three types of output enables used in programmable logic devices; pin controlled, term controlled, and configurable. Each of these output enables, and how it is implemented in ABEL, is discussed in the following paragraphs.

Pin Controlled Output Enable 5.1.1

The pin controlled output enable is illustrated in figure 5-1. Here, the output enable input is a specific device pin; FO is high-impedance for OE, and is active for !OE. The pin controlled output enable is not included in the design equations but should be included in all test vectors. An example of test vectors that include this type of output enable is shown below. These test vectors are for the decimal up/down counter presented in section 2.5. Note that the OE input is included in each vector.

```
test_vectors ( [Clk, OE, Mode ] -> [Q3,Q2,Q1,Q0])
                [ Ck,  0,  Clear] -> S0;
                [ Ck,  0,    Up] -> S1;
                [ Ck,  0,    Up] -> S2;
                [ Ck,  0,    Up] -> S3;
                [ Ck,  0,    Up] -> S4;
                [ Ck,  0,    Up] -> S5;
                [ Ck,  0,    Up] -> S6;
                [ Ck,  0,    Up] -> S7;
                [ Ck,  0,    Up] -> S8;
                [ Ck,  0,    Up] -> S9;
                [ Ck,  0,    Up] -> S0;
                [ Ck,  0,    Up] -> S1;
                [ Ck,  0,  Down] -> S0;
                [ Ck,  0,  Down] -> S9;
                [ Ck,  1,  Down] -> Z;
                [ Ck,  0,  Down] -> S7;
                [ Ck,  0,  Clear] -> S0;
```

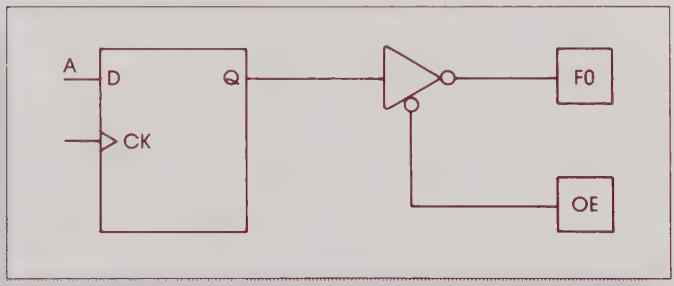


Figure 5-1. Output Enable Controlled by Device Pin

Term Controlled Output Enable 5.1.2

An example of a term controlled output enable is shown in figure 5-2. In this example, the product term that enables pin F0 is A&B. The equation in the ABEL source file would be

```
enable F0 = A&B;
```

For devices that have an active low output enable, a ! can be placed on either side of the equation as follows

```
!enable F0 = x & y;
enable F0 = !(x & y);
```

If the statement

```
enable !F0 = x & y;
```

is used, ABEL ignores the ! and assumes the equation

```
enable F0 = x & y;
```

Another example of the term controlled output enable can be found in section 2.3. Here, the product term inserted in the equation portion of the ABEL source file is

```
enable outputs = enb;
```

This equation defines the "enb" input, previously declared as a pin, as the "enable outputs" function.

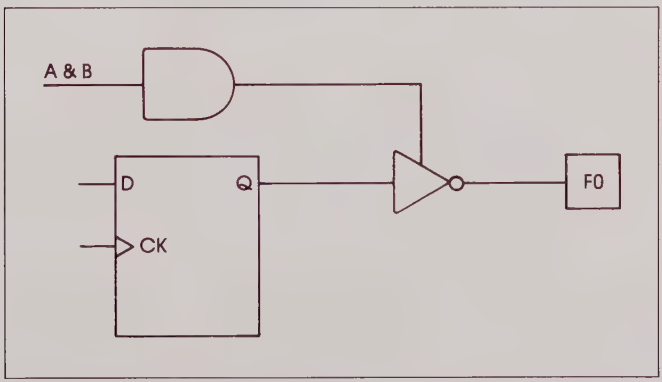


Figure 5-2. Output Enable Controlled by Product Term

Configurable Output Enable 5.1.3

The configurable output enable is one that, through the blowing of a fuse, or fuses, can be made to operate in any of three distinct modes:

- All related outputs enabled regardless of output enable line condition.
- All related outputs are controlled by the output enable line.
- All related outputs are permanently in the high-impedance state (disabled).

Figure 5-3 shows how a pair of fuses determine the configuration of the output enable feature for this type of programmable logic device. The equation

```
ENABLE F0 = 1;
```

blows the upper fuse and leaves the lower fuse left intact. This configuration enables related outputs regardless of the output enable line.

The equation

```
ENABLE F0 = OE;
```

blows the lower fuse and leaves the upper intact. This configuration allows the related outputs to be controlled by the state of the output enable input.

The equation

```
ENABLE F0 = 0;
```

blows both fuses causing the related outputs to be permanently in the high-impedance state.

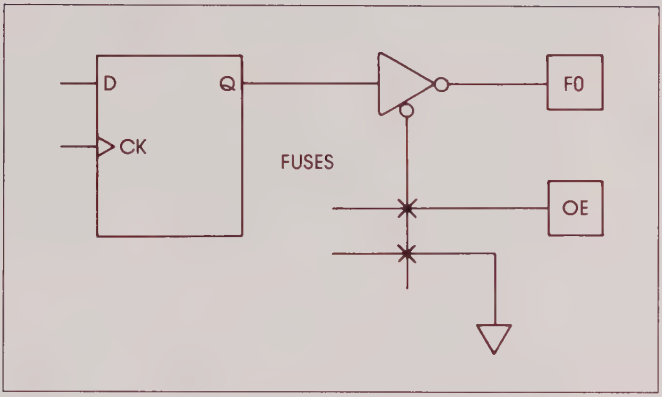


Figure 5-3. Configurable Output Enable

OUTPUT MACRO CELL CONTROL WITH ISTYPE 5.2

The ISTYPE statement, described in the Language Reference Manual, can be used to control the macro cells found in several logic devices, such as the 82S159, the EP300, the P22V10, etc. Macro cells are controllable to several different output configurations with regard to polarity, feedback point, and type, such as shown in the examples of figure 5-4.

Controlling Macro Cell Polarity 5.2.1

Output polarity is controlled by entering "pos" or "neg" in the ISTYPE statment. The ISTYPE statement for Q18 (see listing 5-1) is an example of controlling the output polarity to active low. For an active high output, this statement would be

```
Q18     IsType     'pos,reg,feed_reg';
```

If no polarity is indicated in the ISTYPE statement, a default value of active high is assumed.

The output polarity (active high or active low) of the macro cell device will affect the DeMorgan's equation inversion, and can result an excess of product terms, extended reduction times, and high memory usage. For example, a typical equation, such as

```
Y1  = (A # B) & (C # D) & (E # F);
```

will reduce to the following three product terms for an active low output.

```
!Y1  =  (!A & !B
        # !C & !D
        # !E & !F);
```

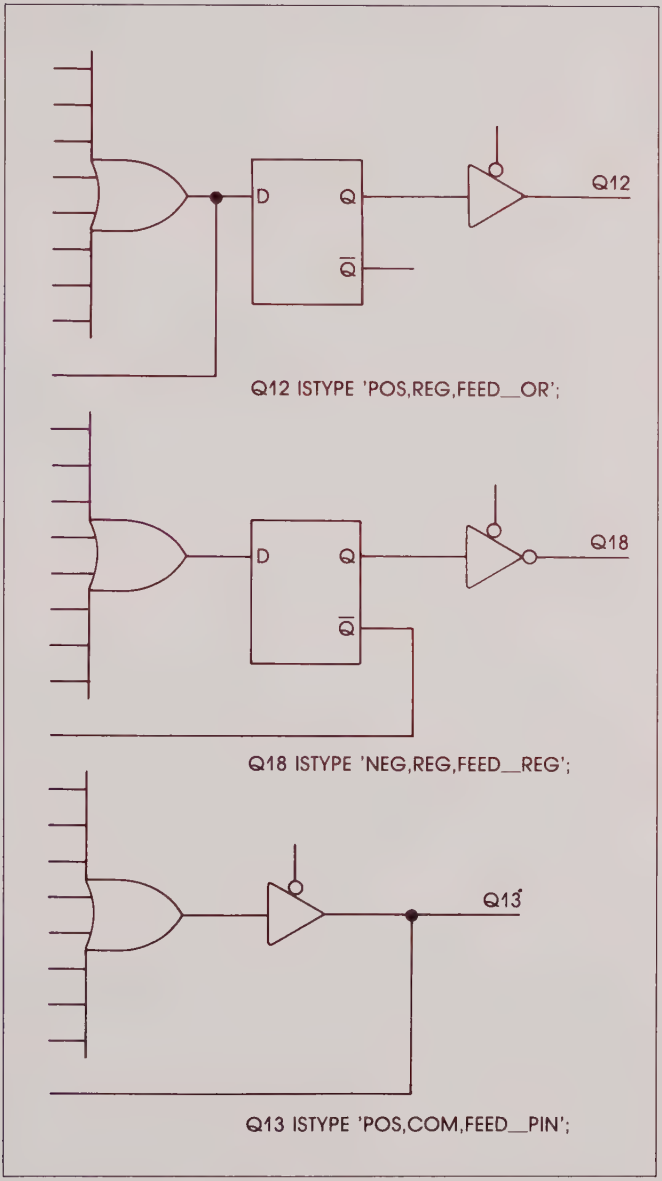


Figure 5-4. Controlling Macro Cells with ISTYPE

However, if an active high output is specified, the same equation requires eight product terms, as follows

```
Y1 = (A & C & E
    #A & C & F
    #A & D & E
    #A & D & F
    #B & C & E
    #B & C & F
    #B & D & E
    #B & D & F);
```

When dealing with programmable polarity devices, where the output can be programmed as active-low or active-high, some consideration should be given to polarity. For most programmable polarity devices, ABEL defaults to an active-high selection when the polarity is not specified. In some devices this can mean a design will not fit into the device (causing a "too many terms for output" error message), although it would if active-low polarity was specified for the outputs.

Controlling Macro Cell Feedback Point 5.2.2

Listing 5-1 also shows that the ISTYPE statement can be used to explicitly state the macro cell configuration, such as in the case of output Q12. Or, the ISTYPE statement can be used to state the feedback point only, leaving the polarity and register/combinatorial selection to be determined by ABEL from the equations. For example, the ISTYPE statement defines the Q14 output as a feedback from the OR function only.

The equation

```
!Q14 := D2 & D3
```

further defines Q14 as an active low, registered output.

```
module altera   flag '-f'
title 'Controlling output macro cells
Brian Durwood     FutureNet - Data I/O     8 Nov 1985'

        alt1    device 'EP300';

        Clk,D2,D3,D4,D5,D6,D7    pin 1,2,3,4,5,6,7;
        D8,D9,D11,Q12,Q13,Q14    pin 8,9,11,12,13,14;
        Q15,Q16,Q17,Q18,Q19      pin 15,16,17,18,19;

"Explicity state macro cell configuration
        Q12              IsType  'pos,reg,feed_or';
        Q13              IsType  'pos,com,feed_pin';
        Q18              IsType  'neg,reg,feed_reg';

"Let equations determine resister/combinatorial and polarity
        Q14              IsType  'feed_or';
        Q15,Q16,Q17      IsType  'feed_pin';

        Ck,X,Z  = .C. , .X., .Z.;

equations
        Q12     := D2 & D3;      "Feedback from the pin
        Q13      = Q12 & D4;

test_vectors    ([Clk,D2,D3,D4] -> [Q12,Q13])
                 [ Ck, 1, 1, 1] -> [ 1 , 1 ];
                 [  0, 1, 0, 1] -> [ 1 , 0 ];
                 [ Ck, 0, 1, 1] -> [ 0 , 0 ];
                 [  0, 1, 1, 1] -> [ 0 , 1 ];

equations
        !Q14    := D2 & D3;      "Registered      Active Low
         Q15     = D3 & D4;      "Combinatorial   Active High

test_vectors    ([Clk,D2,D3,D4] -> [!Q14,Q15])
                 [ Ck, 1, 1, 1] -> [  1 , 1 ];
                 [ Ck, 1, 0, 1] -> [  0 , 0 ];
                 [ Ck, 1, 1, 0] -> [  1 , 0 ];
                 [ Ck, 0, 1, 1] -> [  0 , 1 ];

equations       "bidirectional buffer
        Q16     = Q17;           enable Q16 =  D4;
        Q17     = Q16;           enable Q17 = !D4;

test_vectors    ([D4,Q16,Q17] -> [Q16,Q17])
                 [ 1,  X,  0] -> [  0,  X];
                 [ 1,  X,  1] -> [  1,  X];
                 [ 0,  0,  X] -> [  X,  0];
                 [ 0,  1,  X] -> [  X,  1];

equations
        Q18     :=  D5 & !Q18;
        Q19      =  Clk;

test_vectors    ([Clk,D5] -> [Q18,Q19])
                 [ Ck, 0] -> [  0,  0];
                 [ Ck, 1] -> [  1,  0];
                 [ Ck, 1] -> [  0,  0];
                 [ 0 , 1] -> [  0,  0];
                 [ 1 , 1] -> [  1,  1];
end
```

Listing 5-1. Controlling Macro Cells with ISTYPE Statements

Controlling Macro Cell Type 5.2.3

As shown in listing 5-1, use of : = in the equation causes ABEL to choose a registered output, while = chooses a combinatorial output. As in the case of feedback point and polarity, the ISTYPE statement can be used to explicitly state the macro cell configuration, such as in the case of output Q12.

Controlling Register Type 5.2.4

On devices with programmable register type, such as the 82S159, the register type may be specified by entering "reg_D", "reg_JK", "reg_RS", or "reg_T" in the ISTYPE statement.

NODES 5.3

Nodes are "pseudo-pins"; internal signals that are not accessible on the device's external pins, but that are needed to program otherwise inaccessible fuses. Nodes for specific programmable logic devices are defined by Data I/O and listed in the Device Nodes appendix at the back of the ABEL User's Manual. The listed nodes are known to the ABEL Language Processor and allow you to write equations for certain internal device signals that do not appear as device inputs or outputs. Figure 5-5 is a logic diagram for P22V10 with the node numbers added. (Similar logic diagrams for all devices with node numbers added are provided with the Programmable Logic Device Logic Diagrams, document number 981-0117, included with the ABEL package.)

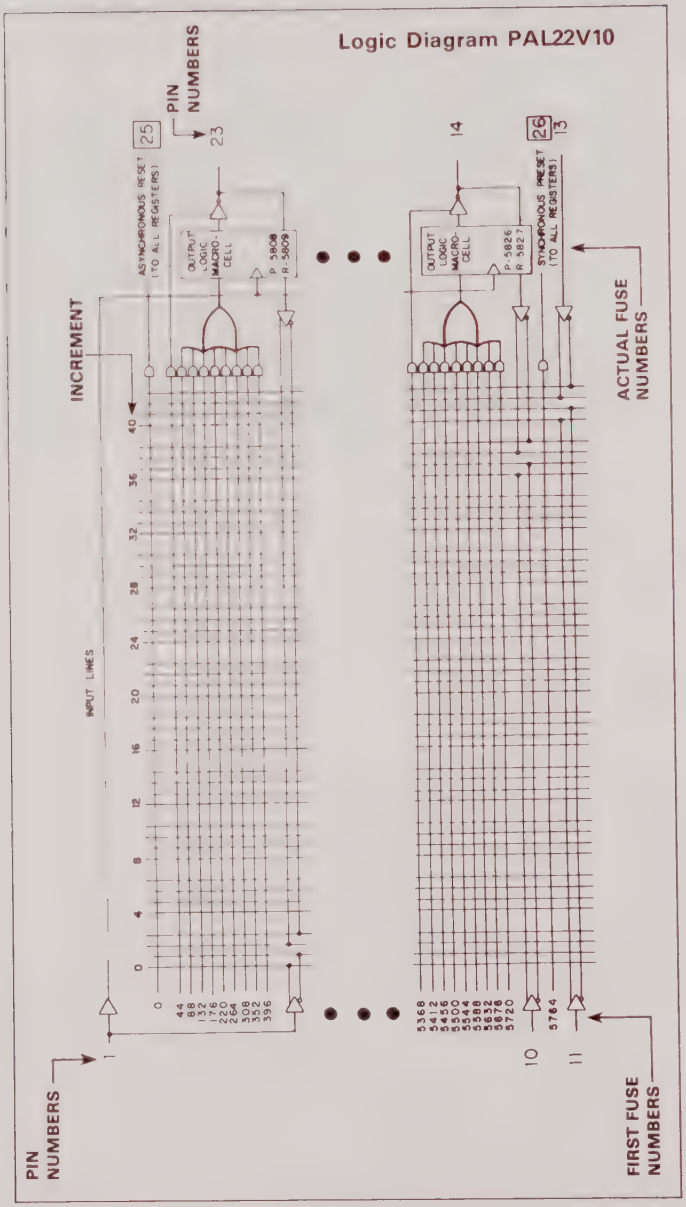


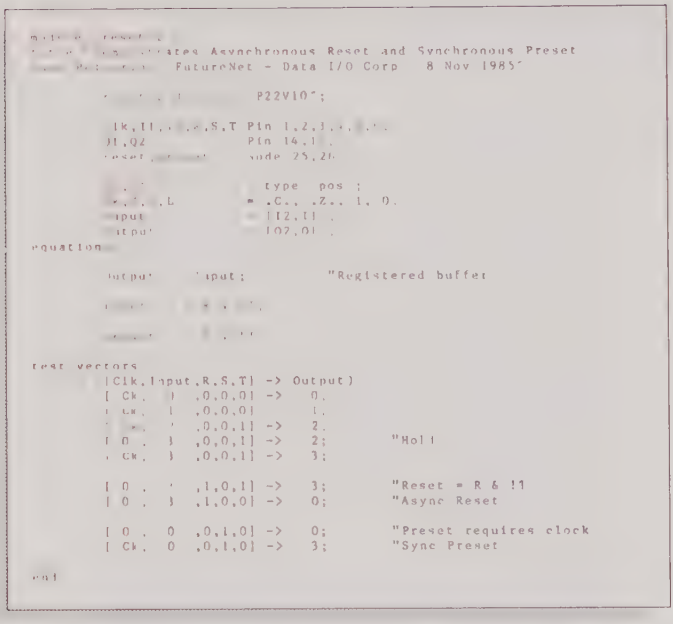
Figure 5-5. Logic Diagram with Node Numbers

Using Node Numbers 5.3.1

In the P22V10, reset and preset functions appear as nodes 25 and 26. As shown in listing 5-2, these nodes can be controlled by product terms placed in the "equations" portion of the ABEL source file. The equations for the nodes are written as though the nodes are actual inputs (or outputs) of the device. The language processor correctly interprets these equations to create the programmer load file.

In listing 5-2, the declaration statements assign the identifiers "reset" and "preset" to nodes 25 and 26 (the reset and preset functions as defined in the Device Nodes appendix). Once declared, the identifier "reset" is equated to pin 4 (R) high and pin 6 (T) low; while "preset" is equated to pin 5 (S) high and pin 6 (T) low. The output (pins 14 and 15) is equated to the input (pins 2 and 3), but registered.

The "reset" equation in listing 5-2 means that the P22V10 device will be asynchronously reset (the P22V10 reset function is asynchronous to the clock input) when pins 4 and 6 are pulled high and low respectively. The "preset" equation means that the P22V10 device will be preset on the occurence of a clock pulse (the P22V10 preset function is synchronous) when pins 5 and 6 are pulled high and low respectively. The test vectors in listing 5-2 verify the reset and preset functions. For a more complete explanation of these test vectors, refer to section 4.7.4.



```
                    Asynchronous Reset and Synchronous Preset
                    FutureNet - Data I/O Corp   8 Nov 1985

                    P22V10;

Clk,I1,  ,S,T Pin 1,2,3,4,
O1,O2         Pin 14,1
                    node 25,26

                    type  pos ;
                  = .C., .Z., 1, 0.
                  [I2,I]
                  [O2,O]

equation

          Output    Input;          "Registered buffer


test_vectors
          ([Clk,Input,R,S,T] -> Output)
          [ Ck,  ] ,0,0,0] ->   0.
          [ Ck,  ] ,0,0,0]      1.
          [ Ck,  ] ,0,0,1] ->   2.
          [ 0 ,  ] ,0,0,1] ->   2;        "Hold
          [ Ck,  ] ,0,0,1] ->   3;

          [ 0 ,  ] ,1,0,1] ->   3;        "Reset = R & !1
          [ 0 ,  ] ,1,0,0] ->   0;        "Async Reset

          [ 0 ,  0 ,0,1,0] ->   0;        "Preset requires clock
          [ Ck,  0 ,0,1,0] ->   3;        "Sync Preset

end
```

Listing 5-2. Using Product Terms to Control, Reset and Preset Nodes

Using Shorthand Notation for Nodes 5.3.2

For many devices, such as those listed in the Device Nodes appendix in the ABEL User's Manual, ABEL does not require that the nodes be declared in declaration statements. For these devices, a shorthand notation is available that allows ABEL to interpret the desired node without a node declarations statement. For example, the preset term (node number 25) for the P22V10 can be simply noted as ".PR" in an equation instead of using an identifier taken from a node declaration statement. Similarly, the reset term can be noted as ".RE". Other devices may also use this shorthand notation; these devices are listed in the Device Nodes appendix of the User's Guide.

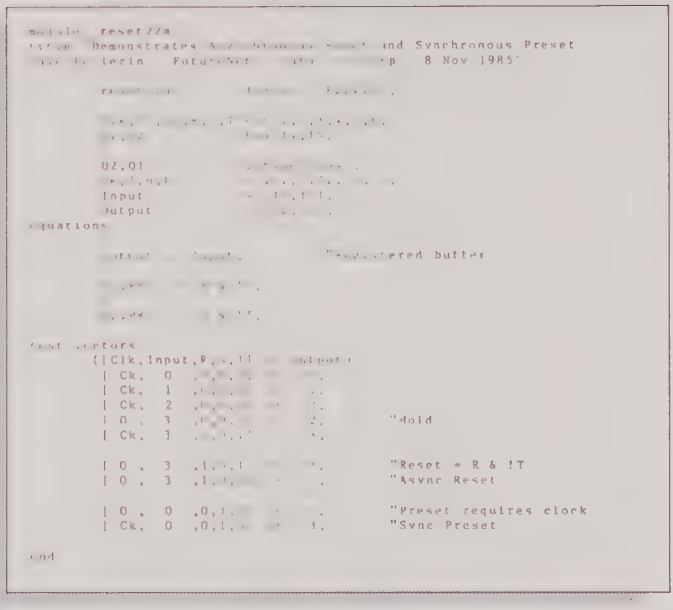
The shorthand notation (also described in section 5.4.5) eliminates the need to enter node declaration for these nodes. To specify the preset and reset nodes in the equations without declaring, the shorthand notation given in the Device Nodes appendix is used directly in the equations. For example, the reset input of the flip-flop associated with the declared Q2 output is noted as Q2.RE, as shown in listing 5-3. Similarly, the preset input for the Q2 flip-flop is noted as Q2.PR. Listing 5-3 shows the equations are written as

```
Q2.RE  = R & !T
Q2.PR  = S & !T
```

instead of

```
reset  = R & !T
preset = S & !T
```

as in listing 5-2.



```
Module   reset22a
Title    Demonstrates A[illegible] and Synchronous Preset
Date     8 Nov 1985
[illegible]

02.01
[illegible]
Input
Output
Equations

[illegible]                    "[illegible]ered butter

[illegible]

[illegible]

Test_vectors
        ([Clk,Input,P,[illegible]] -> [Output]
         [ Ck,  0  ,[illegible]]
         [ Ck,  1  ,[illegible]]
         [ Ck,  2  ,[illegible]]
         [ 0 ,  3  ,[illegible]]        "Hold
         [ Ck,  3  ,[illegible]]

         [ 0 ,  3  ,[illegible]]        "Reset = R & !T
         [ 0 ,  3  ,[illegible]]        "Async Reset

         [ 0 ,  0  ,[illegible]]        "Preset requires clock
         [ Ck,  0  ,[illegible]]        "Sync Preset

end
```

Listing 5-3. Using Shorthand Notation to Control Nodes

USING FLIP-FLOPS (MACRO CELL DEVICES) 5.4

Devices that contain macro cells allow you to select from several output functions to provide flexibility in logic design. A common selection that can be made in a macro cell is the configuration of the flip-flops to accomodate specific logic functions. To illustrate the use of ABEL in the control of these flip-flops, a design example is described in the paragraphs that follow. The design example is a four-bit shifter/counter implemented in a F82S159 field programmable logic sequencer. Figure 5-6 shows a simplified block diagram of the design example. The design has four inputs and four outputs. Inputs JKD, and I0-I3 provide control of the counter and determine whether the counter is to count up or down, shift left or right, or be cleared. (The counter in this design has no loadable inputs.) Outputs F0 - F3 reflect the current count, the shifted outputs, or the cleared outputs, depending on the mode of the shifter/counter.

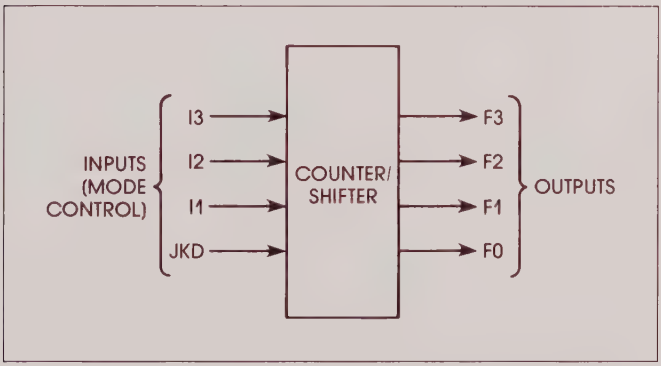


Figure 5-6. Counter/Shifter Block Diagram

The 82S159 Logic Diagram 5.4.1

Figure 5-7 is a logic diagram of the 82S159 FPLS and shows the details of the eight registered-output macro cells. The shifter/counter design used in this example employs the lower four of these outputs. Note that the registers in this device are dynamically programmable as either J-K or D-type flip-flops by means of the FC line. Dynamic programming of the flip-flops is possible as long as the fuses on the M0 through M3 lines remain intact and are not blown during the design process. (If any of the Mx fuses are blown, the associated register becomes a J-K flip-flop permanently. See section 5.4.3.)

The dynamic programming of the output registers allows the use of J-K flip-flops when the counter mode is selected, and D-type when the shift function is selected. This is ideal since J-Ks are better suited to counters and D-types are better suited to shifters.

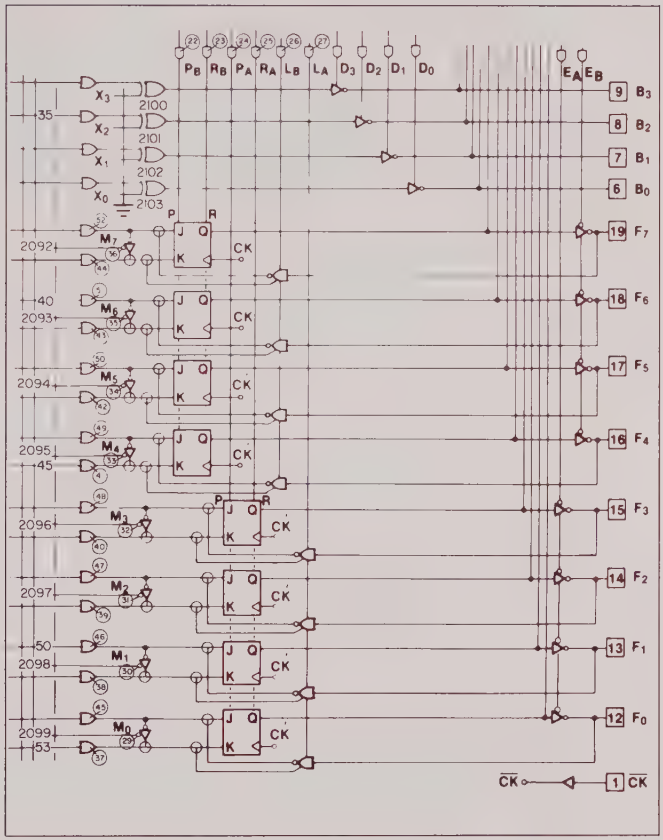


Figure 5-7. Logic Diagram of the 82S159 FPLS (partial)

Examination of the Source File 5.4.2

The source file for the shifter/counter is shown in listing 5-4. Pin declarations are made to define inputs and outputs, and node declarations are made to declare the flip-flop control (FC) node (to provide the dynamic programming of the flip-flops) and the RA node (to provide the clear function for the counter).

The special constants definitions define the terms "INTACT" and "OPEN". These are terms that can be used in the equations to leave specified fuses intact or blow them. "INTACT" is specified as 1111 while "OPEN" is defined as 0000.

Sets are used to define the mode control lines and the configuration of these lines for each of the shifter/counter modes. Inputs JKD, and I1 through I3 are defined as "mode". Under "mode", each shifter/counter function is defined as specific inputs to JKD and I1 through I3 (pins 2 through 5). The four outputs (F0-F3) are defined as "count", which is used in the first equation to configure the output enable input. (Refer also to section 5.1.3.)

The second equation causes the fuses that control the mode of the flip-flops (J-K or D-type) to remain intact. The 82S159 defaults to the "INTACT" condition, so the second equation in the example is not really necessary. It is provided here to illustrate the method of blowing the fuses associated with these nodes, or leaving them intact. Refer to Programming the Flip-Flop Mode Control Fuses, 5.4.3 for information on programming these fuses.

The next equation defines the FC (J-K/D-type flip-flop function control) line as equal to input line JKD (pin 2). The remaining equations define the operation of the shifter/counter. Some rules apply to these latter equations and are explained in the following sections 5.4.4 and 5.4.5. The final equation defines the reset input to the flip-flops (RA, node 25) as equal to input line I3 pulled high. That is,

RA = (mode == Clear)

which is equal to

RA = [I3,I2,I1,JKD] == [1,X,X,X]
= I3

Programming the Flip-Flop Mode Control Fuses 5.4.3

If the M0 through M3 nodes of the 82S159 were declared in the declaration of the module, then the equation

```
[M3,M2,M1,M0] = INTACT;
```

could be written to program the flip-flop mode control fuses (assuming INTACT is defined as a special constant). However, the equation

```
[F3.M,F2.M,F1.M,F0.M] = INTACT
```

is used in listing 5-4 to program the flip-flop mode control fuses. This form of the equation uses a notation for the M (function control) node of the flip-flops that does not require that nodes be declared. The notation "F3.M" indicates the M node of the register associated with the F3 output; "F2.M" for the M node of the F2 output, etc.

If it were desired to permanently configure one or more of the flip-flops to the J-K function, it would be necessary to blow the M node fuses associated with the flip-flops. If in listing 5-4, it was desired to configure the flip-flops designated F0 through F3 permanently to J-K types, you could use the equation:

```
[F3.M,F2.M,F1.M,F0.M] = 0
```

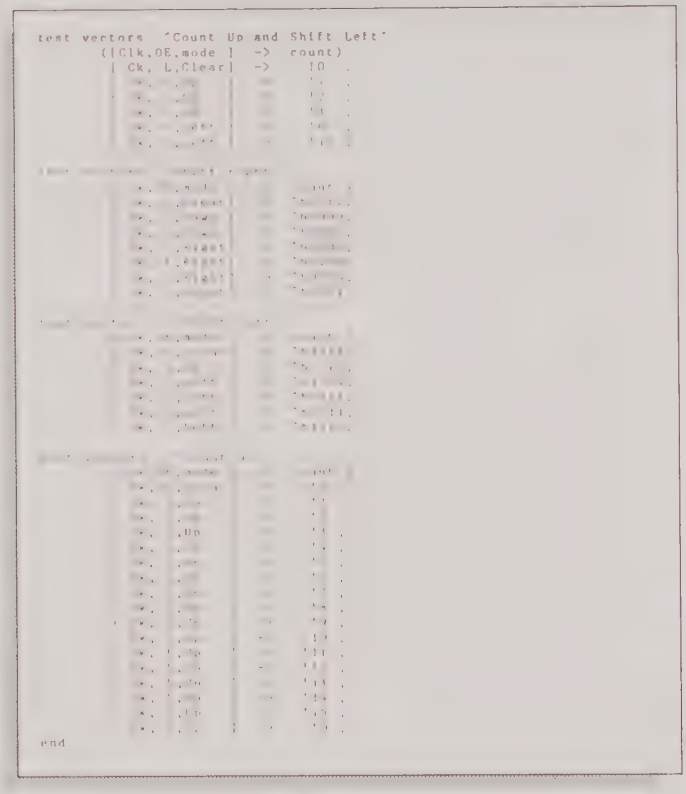
You could also program the flip-flop mode control fuses to a desired bit pattern by writing an equations that specifies the bit pattern. For example, the equation

```
[F3.M,F2.M,F1.M,F0.M] = ^b0110
```

sets flip-flops F3 and F0 permanently to J-K types while leaving F2 and F1 as controllable.

```
module shiftcnt   flag '-t1'
title 'universial counter / shift register
Bjorn Benson    Data I/O Corp    9 Mar 1984'

        IFL4    device 'F82S159';

        Clk,JKD,I1,I2,I3,OE     pin 1,2,3,4,5,11;
        F0,F1,F2,F3             pin 12,13,14,15;
        FC,RA                   node 21,25;

        H,L,X,Z,Ck      = 1,0,.X.,.Z.,.C.;
        INTACT,OPEN     = ^hFF,0;

" Define counter/shift modes
        mode    = [I3,I2,I1,JKD];               " [I3,I2,I1,JKD]
        Down    = [ 0, 1, 0, 1 ];       Left    = [ 0, 1, 0, 0 ];
        Up      = [ 0, 0, 1, 1 ];       Right   = [ 0, 0, 1, 0 ];
        Clear   = [ 1, X, X, X ];

" Group register outputs into sets
        count           = [F3,F2,F1,F0];

equations
        ENABLE  count = !OE;

        [F3.M,F2.M,F1.M,F0.M] = INTACT;
        FC = JKD;                       "JK when High, D when Low

" Count Down     JK (T) Flip/Flop
        !F0     := (mode == Down)  ;
         F0.K    = (mode == Down)  ;
        !F1     := (mode == Down)  &                            F0;
         F1.K    = (mode == Down)  &                            F0;
        !F2     := (mode == Down)  &                    F1 &    F0;
         F2.K    = (mode == Down)  &                    F1 &    F0;
        !F3     := (mode == Down)  &            F2 &    F1 &    F0;
         F3.K    = (mode == Down)  &            F2 &    F1 &    F0;

" Count Up       JK (T) Flip/Flop
        [!F0,F0.K]      := (mode == Up)     ;                                   !F0;
        [!F1,F1.K]      := (mode == Up)     &                           !F1 & !F0;
        [!F2,F2.K]      := (mode == Up)     &                   !F2 & !F1 & !F0;
        [!F3,F3.K]      := (mode == Up)     &           !F2 & !F1 & !F0;

" Shift Left     D Flip/Flop
        !F0             := (mode == Left)  & !F3;
        !F1             := (mode == Left)  &                                    'F0;
        !F2             := (mode == Left)  &                            !F1;
        !F3             := (mode == Left)  &            !F2;

" Shift Right    D Flip/Flop
        !F0             := (mode == Right) &                            !F1;
        !F1             := (mode == Right) &            !F2;
        !F2             := (mode == Right) & !F3;
        !F3             := (mode == Right) &                                    !F0;

" Clear
        RA      = (mode == Clear);

@page
```

Listing 5-4. A Shifter/Counter in an F82S159



Listing 5-4. *(Continued)*

Combining Equations 5.4.4

The equations in the "Count Down" portion of the source file can be written as shown, or you can combine pairs of equations that are identical on the righthand side. That is, the two equations

```
!F0 := (mode == Down);
F0.K = (mode == Down);
```

may be written as the single equation

```
[!F0,F0.K] := (mode == Down);
```

The latter equation shows how the two equations have been ORed into a single equation without altering the specified design of the device. Similarly,

```
!F2 := (mode == Down) & F1 & F0;
F2.K = (mode == Down) & F1 & F0;
```

may be written as

```
[!F2,F2.K] := (mode == Down) & F1 & F0;
```

Specifying the Flip-Flop Inputs 5.4.5

The inputs of the flip-flops, such as those in the equations portion of listing 5-4, are identified in either of two ways: 1) by means of node numbers and identifiers defined in node declarations statements, or 2) by a shorthand notation that associates the inputs with their output identifiers. To specify the flip-flop inputs using their node numbers (listed in the Device Nodes appendix in the User's Guide) you must first declare the J and K inputs in the declaration portion of the module, and then use the declared identifier in the equations. This method is used exclusively in earlier versions of ABEL.

To specify the flip-flop inputs without declaring them as nodes, the shorthand notation given in the Device Nodes appendix is used directly in the equations. For example, the J and K inputs of the flip-flop associated with the F1 output of the 82S159 may be noted as F1.J and F1.K, as shown in the upper portion of figure 5-8. The inputs for other flip-flops in the design may be noted in a similar manner as shown below.

```
[F1.J,F1.K] := (mode == Down)
[F2.J,F2.K] := (mode == Down) & F1 & F0;
[F3.J,F3.K] := (mode == Down) & F2 & F1 & F0;
```

This method of flip-flop input notation, shown in the following equations, eliminates the need to declare the nodes for the flip-flop inputs.

Instead of being noted as described above, the J input (but not the K input) can also be noted in terms of the registered output, as shown in the lower portion of figure 5-8. This is due to the fact that whatever logic level is placed on the J input of the register appears at its output. (In figure 5-8, the complement of the output is used due to the inverter in the 82S159 output.) For example, the equations

```
[F1.J,F1.K] := (mode == Down)

[F2.J,F2.K] := (mode == Down) & F1 & F0;

[F3.J,F3.K] := (mode == Down) & F2 & F1 & F0;
```

could be written as

```
[!F1,F1.K] := (mode == Down)
[!F2,F2.K] := (mode == Down) & F1 & F0;
[!F3,F3.K] := (mode == Down) & F2 & F1 & F0;
```

If a J input is noted in one equation as "F1.J" and in another as "!F1" (the name of the registered output), an inconsistency is introduced into the design equations that will result in an error. The equations

```
[F2.J,F2.K] := (mode == Down) & F1 & F0;
[F3.J,F3.K] := (mode == Down) & F2 & F1 & F0;
```

consistently define the J inputs, however the equations

```
[!F2,F2.K] := (mode == Down) & F1 & F0;
[F3.J,F3.K] := (mode == Down) & F2 & F1 & F0;
```

note the J input in the two different ways.

In the last example, a J input is noted in terms of its J input node and another as the inverse of its registered output. This type of inconsistency will result in a fusemap error when the design is processed by ABEL. It is important to decide on the form used to specify the J inputs and use only that form.

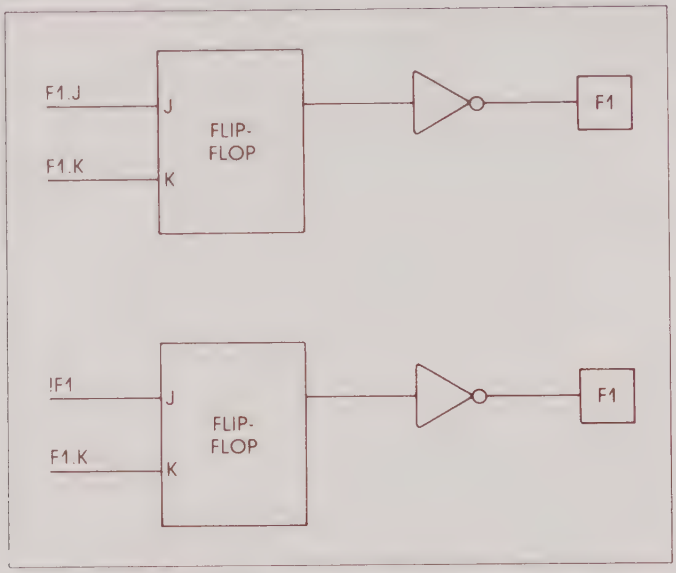


Figure 5-8. Flip-Flop Input Notation Examples

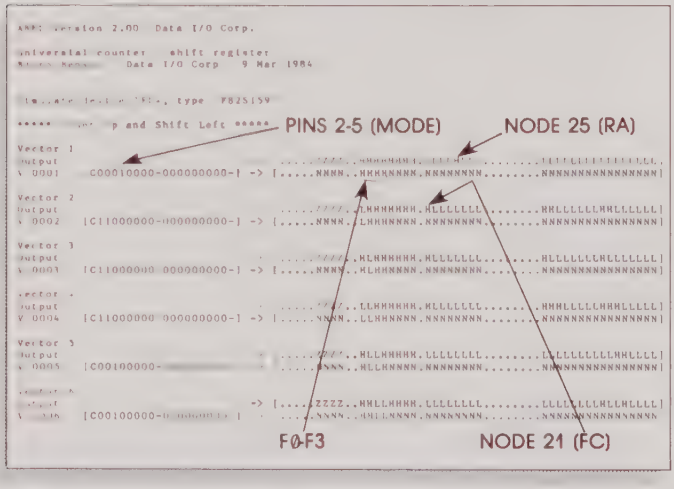
Test Vectors 5.4.6

The test vectors in listing 5-4 are divided into four groups for the sake of functional clarity. The first group simply clears the device and then makes it count up three increments and then shift left twice. This test is provided to show the operation of the design when it is run through the ABEL simulator and then displayed as shown in listing 5-5 (described in text which follows). The second group of test vectors tests the shift right function by first clearing the counter to all active-high outputs and then decrementing it to a single active-high output. The last four vectors shift the active-high bit across the outputs. The third group of vectors operates in a similar manner to test the shift left function. The last group of test vectors test the counter from state 0 through state 15 and then wraparound back to state 0.

The first group of test vectors provide a test, which when the source file of listing 5-4 is run through the ABEL simulator, generate the trace information shown in listing 5-5. (Listing 5-5 is a partial printout of the "shftcnt.SIM" file generated by the SIMULATE element of the ABEL language processor. Listing 5-5 shows the SIMULATE results of the first six test vectors contained in listing 5-4.) The input side of the first test vector shows that pin 5 is high to clear the counter. On the output side of the vector, node 25 (the RA reset register bank A function) is high to clear the counter registers; outputs F0 through F3 show that the counter outputs are all high, indicating the reset condition (the 82S159 has active-low register outputs).

The second vector of listing 5-5 shows the change in the mode inputs to the count-up function, i.e., pins 4 and 5 at logic high. On the output side of the vector, the F0 pin (pin 12) is active low and the FC line (node 21) is now active to place the flip-flops in the J-K mode as described in section 5.4.1. The third and fourth vectors show that the mode selected on the input side remains the count-up function while the F0 through F3 outputs increment in a binary fashion to a count of three (decimal).

The fifth vector shows that the mode is changed to the shift-left function and the F0 through F3 outputs shift as a result to provide an output of six (decimal). The sixth vector provides a second shift-left function, resulting in a shifter output equal to 12 (decimal) or C (hexadecimal).



Listing 5-5. SIMULATE Output (Partial) for the Shifter/Counter

USING TRANSITION EQUATIONS 5.5

Transition equations may be used to express the design of counters and state machines in some devices that contain J-K and/or R-S flip-flops. A transition equation expresses a state of the circuit as a variation of, or an adjustment to, the previous state. This type of equation eliminates the need to specify each node of the circuit, but only those that require a transition to the opposite state.

An example of transition equations usage is shown in listing 5-5, a source file for a decade counter having a single (clock) input and a single latched output. The purpose of this counter is to divide the clock input by a factor of ten and generate a 50% duty-cycle squarewave output. The device uses for this design is an 82S105 FPLS. In addition to its registered outputs, this device contains a set of "buried" (or feedback) registers whose outputs are fed back to the product term inputs.

Before examining the equations in listing 5-6, note that the decade counter design is shown in figure 5-9. Figure 5-9 is an abbreviated logic diagram of the 82S105 that shows the "buried" (no output pins) registers arranged as a counter having feedback paths back to the inputs by means of product terms. The counter registers are designated P0 through P3, as declared in the node declaration statements of the source file in listing 5-6. These node designations can then be used in the equations. The output register is declared as F0.

Node 49, the complement array feedback, is declared (as "COMP") so that it can be entered into each of the equations. In this design, the complement array feedback is used to wrap the counter back around to zero from state nine, and also to reset it to zero in the event an illegal counter state is encountered. Any illegal state (and also state 9) will result the absence of an active product term to hold node 49 at a logic low. When node 49 is low, figure 5-9 shows that product term 9 resets each of the feedback registers so that the counter is set to state zero. (To simplify the following description of the equations in listing 5-6, node 49 and the complement array feedback are temporarily ignored.)

The first equation states that the F0 (output) register is set (to provide the counter output) and the P0 register is set when registers P0, P1, P2, and P3 are all reset (counter at state zero) and the clear input is low. Figure 5-9 shows how the fuses are blown to fullfill this equation; the complemented outputs of the registers (with the clear input low) form product term 0. Product term 0 sets register P0 to increment the decade counter to state 1, and sets register F0 to provide an output at pin 18.

The second equation performs a transition from state 1 to state 2 by setting the P1 register and resetting the P0 register. (The .R shorthand notation described in section 5.3.2 is used to define the reset input of the registers.) In state 2, the F0 register remains set, maintaining the high output. The third equation again sets the P0 register to achieve state 3 (P0 and P1 both set), while the fourth resets P0 and P1, and sets P2 for state 4, and so on.

Wraparound of the counter from state 9 to state 0 is achieved by means of the complement array node (node 49). The last equation defines state 0 (P3, P2, P1, and P0 all reset) as equal to !COMP, that is, node 49 at a logic low. When this equation is processed by ABEL, the fuses are blown as indicated in figure 5-9. Figure 5-9 shows that state 9 (P0 and P3 set) provides no product term to pull node 49 high. As a result, the !COMP signal is true to generate product term 9 and reset all the "buried" registers to zero.

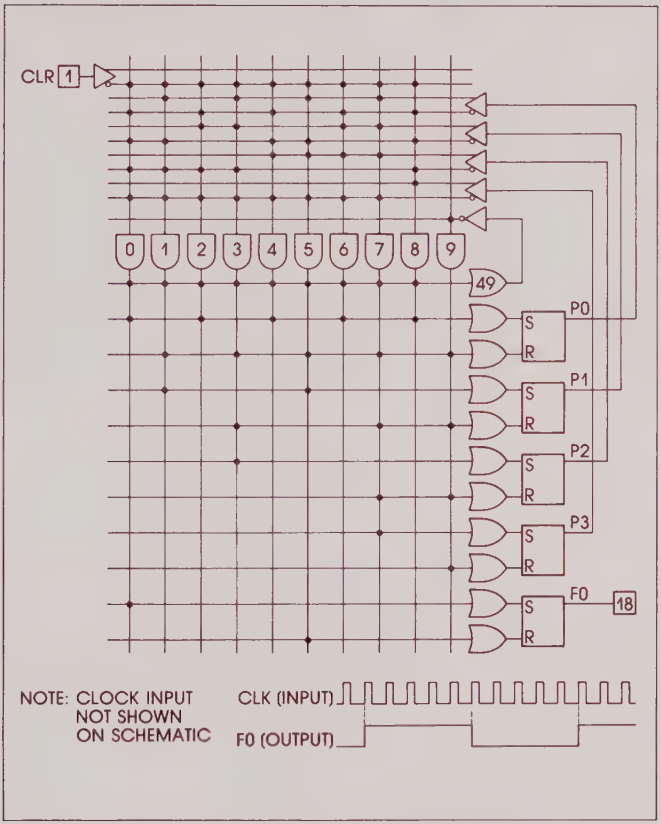


Figure 5-9. Abbreviated 82S105 Schematic

ADVANCED DESIGNS 6

This section contains advanced logic designs described with ABEL™. These designs build upon examples and concepts presented in the earlier sections of this manual. Two advanced designs are presented, a blackjack machine and a 6809 system controller system. Each design is the combination of more than one more basic logic design. Design specification, methods, and complete source files are given for all parts of each example. Table 6–1 lists, for each example, the logic designs which comprise that example.

Table 6–1. Logic Designs Contained Within Examples

Advanced Design	Contains
Blackjack Machine	Multiplexer 5–bit adder Binary to 7–segment decoder Specialized 7–segment decoder State machine controller
6809 System Controller	Address decoder Address multiplexers (2)

BLACKJACK MACHINE 6.1

This describes a classic blackjack machine based on C.R. Clare's design in *Designing Logic Systems Using State Machines* (McGraw Hill, 1972). The machine consists of a card reader and internal logic to play a hand. The dealer's strategy is played to determine whether another card should be drawn: if the total of the drawn cards is less than 17, another is drawn; no cards are drawn if the count is 17 or greater. The machine accounts for the two possible different values of aces depending on the card count total. Status is shown on two LEDs that either show the count total, or symbols for "hit," "stand," or "bust." Figure 6-1 shows a stylized drawing of the blackjack machine.

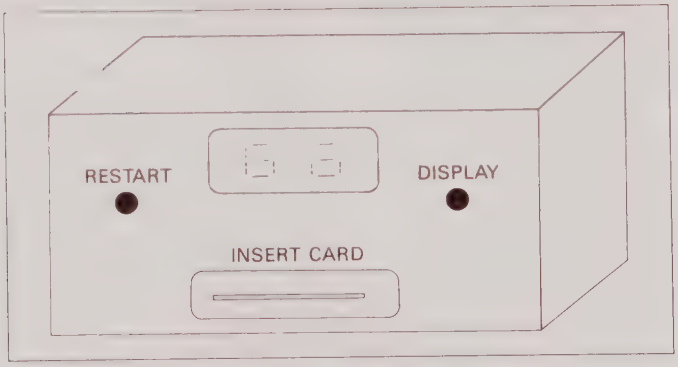


Figure 6-1. Blackjack Machine

Operation of the machine is as follows: A card is inserted into the card reader. The value is read and added to the values of any previously read cards. Values are determined as in normal blackjack: face cards are valued at 10, non-face cards are worth face value, aces are valued at 1 or 11 as necessary or required. If the total

count is less than 17, the machine displays an "H" for "hit," indicating that another card should be inserted in the reader. If the value is equal to or greater than 17 but less than 22, the machine indicates that it will stand by displaying "S". If the total value is greater than 22, the machine has "busted" and indicates this with a "b" on the display. The current count (score) can be displayed at any time by depressing the display button on the machine.

The following text describes the internal logic design necessary to keep the card count, to control the play sequence, and to show the count or state on the LEDs. Neither the card reader nor the physical design is discussed here. It is assumed that the card reader provides a binary value representative of the card read.

Design Specification

The blackjack machine is implemented with five programmable logic devices. The device types and their functions are shown in table 6–2.

Table 6–2. Programmable Devices and Their Function in the Blackjack Machine

P16HD8	Card selector: new card or ace
F82S159	5-bit adder
F82S153	Low-order LED decoder
F82S153	High-order LED decoder
P16R6	State machine controller

Figure 6-2 shows a schematic for the blackjack machine logic. The design consists of a 5-bit adder to total the value of read cards, two 7-segment display decoders to provide the LED output, a state machine controller that controls operation of the machine, and a card selector. The card selector is a 4-to-1 multiplexer that selects either a new card from the reader or one of three internally programmed values (10, -10, 31).

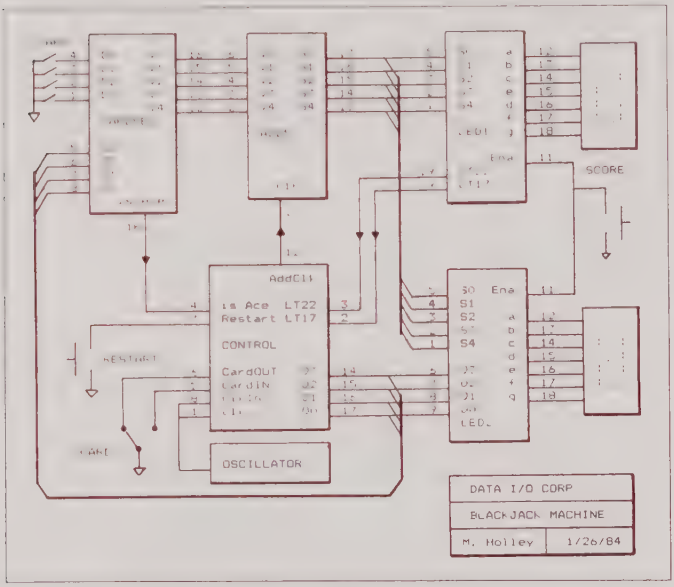


Figure 6-2. Schematic: Blackjack Machine

The design has eight inputs, four of which are the binary encoded card values, *D0-D3*. The remaining four inputs are signals that indicate that the machine is to be restarted, that a card is in the reader, that no card is in the reader, or whether the score or state (hit/stand/bust) is to be displayed. These inputs are *RESTART*, *CardIn*, *CardOut*, and *ena*, respectively. *CardIn* and *CardOut* are provided by the card reader. *RESTART* and *ena* are provided by switches on the exterior of the machine.

LED1 is the seven-segment decoder for the low-order display and provides the modulus 10 of the score to the low-order LED. The decoder must be active low to drive the common-anode LED. When *ena* is pulled low by depression of the display switch on the blackjack machine, the score is displayed on the LED. When *ena* is high (switch not depressed), the outputs of LED1 are all at high impedance and the display is blank.

LED2 is a seven-segment decoder and multiplexer for the high-order display. LED2 displays the state (hit, stand, or bust) of the blackjack machine when *ena* is high. When the display switch on the blackjack machine is depressed, thus pulling *ena* low, the high-order decimal digit of the score is displayed. This digit is obtained through integer division of the score by ten.

ADD5 is a five-bit binary ripple adder that adds the current input from the card selector to the current count. ADD5 is clocked by a signal from the state machine controller. The outputs from ADD5 are provided to both of the display decoders so that they may display the current score. ADD5 is implemented on an F82S159. The F82S159 is a registered IFL with internal feedback, so the current count is always available for adding. The adder is cleared when cleared inputs are received from the card selector (all bits high).

CARDSEL and CONTROL work together to control operation of the blackjack machine and to provide the correct input to the adder. CARDSEL is a four-bit, 4-to-1 multiplexer that selects either the value provided by the card reader or one of three internally programmed values (10, -10, 31) and passes it to the adder. CARDSEL also provides an output that goes high if the value from the card reader is in the correct range, 1 to 10. CONTROL is a state machine controller that controls state transitions and output to the LED display decoders and to CARDSEL. CONTROL is implemented on a P16R6 device with the ABEL™ state diagram structure. The state machine has ten states, as described in table 6-3.

Table 6-3. States of the Blackjack State Machine

State	Description
ShowHit	Indicate that another card is needed. High-order display shows "H".
AddCard	Add the value present at the adder input to the current count.
Add10	Send the value 10 to the card selector to be added to the current count, in effect giving an ace a value of 11.
Wait	Wait until a card is taken out of the reader.
Test17	Test the current count to see if it is less than 17.
Test22	Test the current count to see if it is less than 22.
Sub10	Send the twos complement of 10 to the card selector to be added to the current count, in effect subtracting 10 and restoring an ace to 1 after it was an 11.
ShowBust	Display a "b" on the high-order display to indicate a bust.
ShowStand	Display an "S" on the high-order display to indicate that no more cards are to be drawn.
Clear	Clear the state machine, the adder, and the displays.

Figure 6-3 shows the pictorial state diagram for the controller, indicating state transitions and the conditions causing those transitions. Transitions are shown by arrows, and the conditions causing the transitions are written alongside the arrow.

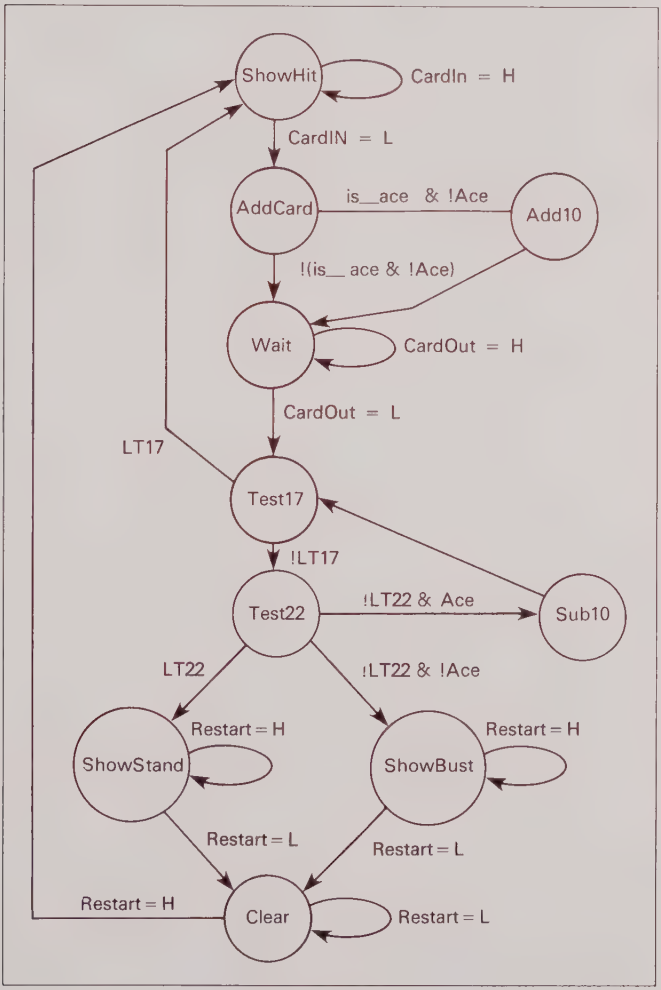


Figure 6-3. Pictorial State Diagram: Blackjack Machine

Operation of the state machine proceeds as follows if no aces are drawn: If a card is needed from the reader, the state machine goes to state *ShowHit*. When *CardIn* goes low, meaning that a card has been read, a transition to state *AddCard* is made. The card value is added to the current score or count value. The machine goes to *Wait* state until the card is withdrawn from the reader. The machine goes to *Test17* state. If the score is less than 17, another card is drawn. If the score is greater than or equal to 17, the machine goes to state *Test 22*. If the score is less than 22, the machine goes to the *ShowStand* state. If the score is 22 or greater, a transition is made to the *ShowBust* state. In either the *ShowStand* or *ShowBust* state, a transition is made to *Clear* when *Restart* goes low, and the state register and adder are cleared. When *Restart* goes back high, the state machine returns to *ShowHit* state and the cycle begins again.

Operation of the state machine when an ace is drawn is essentially the same. A card is drawn and the score is added. If the card is an ace and no ace has been drawn previously, the state machine goes to state *Add10*, and ten is added to the count, in effect making the ace an 11. Transitions to and from *Test17* and *Test22* proceed as before. However, if the score exceeds 21, and an ace has been set to 11, the state machine goes to state *Sub10*, 10 is subtracted from the score, and the state machine goes to state *Test17*.

The current state is always present at the CARDSEL inputs. When the state machine is in state *AddCard*, CARDSEL selects the input from the card reader and passes it to the adder. When the state machine is in states *Add10*, *Sub10*, or *Clear*, CARDSEL selects one of the fixed values 10 (*Add10*), –10 (*Sub10*), or 31 (*Clear*) and passes it to the adder.

Design Method

Each of the five major functions in the blackjack machine is implemented in a separate device and described in a separate module. These modules can be combined and processed as a whole by the language processor, or they can be processed separately. In either case, one JEDEC-format programmer load file is created for each device. The implementation of each function is now discussed.

Several of the different modules needed to describe the blackjack machine must refer to the machine's ten possible states. To simplify the modules, a file containing constant declarations for each of the states was created and is included (using the @INCLUDE directive) in modules where needed. Listing 6-1 shows the file named BJSTATE.TAB, which contains the constant declarations for the states.

```
" State assignments for BlackJack machine  18 Jan 1984

        Clear           = ^b1111;
        ShowHit         = ^b1110;
        AddCard         = ^b1100;
        Add10           = ^b1000;
        Wait            = ^b1010;
        Test17          = ^b0010;
        Test22          = ^b0011;
        ShowStand       = ^b0111;
        ShowBust        = ^b1011;
        Sub10           = ^b0001;
```

Listing 6-1. State Declarations. This file is included where needed in modules with the @INCLUDE directive.

The seven-segment decoder for the low-order LED display is implemented in an F82S153 IFL. The source file describing the decoder is shown in listing 6-2. The score input, *S0-S4*, is decoded to the display segments, *a-g*. Two additional signals, *LT22* and *LT17*, are assigned to output pins 19 and 9. The score inputs and the LED segment outputs are collected into sets named *score* and *led*. *ON* and *OFF* are declared as constants with *ON* being 0, corresponding to a common-anode LED. The set of segment outputs required for each digit that the LED must display are also grouped into sets so that the correct set of outputs can be referred to by name.

The logic is described in an equations section and in a truth table. The equations define the logic function for the LED enable, *LT22*, and *LT17*. The truth table describes the input to output decode function and employs the @CONST, @REPEAT and @EXPR directives to create the truth table. This saves having to fill out the truth table for every possible score. The test vectors are described using the same directives plus a few additional vectors to test the enable line.

```
module  led1
title 'Binary to seven segment display decoder for Blackjack machine
Michael Holley & Walter Bright   Data I/O Corp     24 Feb 1984'
"
"           a        binary-to-seven-segment decimal decoder
"          ---
"        f | | b
"          ---              segment identification
"        e | | c
"          ---
"  ---           ---     ---           ---     ---     ---     ---     ---
" |   |       |     |   |   |   |   |   |   |       |   |   |   |   |   |
"  ---           ---     ---           ---     ---     ---     ---     ---
" |   |       |     |   |   |   |   |   |   |       |   |   |   |   |   |
"  ---           ---     ---           ---     ---     ---     ---     ---

        BJ4A    device  'F82S153';

        S4,S3,S2,S1,S0  pin 1,2,3,4,5;              score=[S4,S3,S2,S1,S0];
        a,b,c,d,e,f,g   pin 12,13,14,15,16,17,18;   led=[a,b,c,d,e,f,g];
        LT22,LT17,ena   pin 19,9,11;

        ON,OFF  = 0,1;                  " for common anode leds
        L,H,Z   = 0,1,.Z.;

        "input                      outputs
        "               a    b    c    d    e    f    g
        digit0  =   [ ON,  ON,  ON,  ON,  ON,  ON, OFF] ;
        digit1  =   [OFF,  ON,  ON, OFF, OFF, OFF, OFF] ;
        digit2  =   [ ON,  ON, OFF,  ON,  ON, OFF,  ON] ;
        digit3  =   [ ON,  ON,  ON,  ON, OFF, OFF,  ON] ;
        digit4  =   [OFF,  ON,  ON, OFF, OFF,  ON,  ON] ;
        digit5  =   [ ON, OFF,  ON,  ON, OFF,  ON,  ON] ;
        digit6  =   [ ON, OFF,  ON,  ON,  ON,  ON,  ON] ;
        digit7  =   [ ON,  ON,  ON, OFF, OFF, OFF, OFF] ;
        digit8  =   [ ON,  ON,  ON,  ON,  ON,  ON,  ON] ;
        digit9  =   [ ON,  ON,  ON,  ON, OFF,  ON,  ON] ;

        HighZ   macro  { [  Z,   Z,   Z,   Z,   Z,   Z,   Z] };

equations
        LT22            = (score < 22);         "Bust
        LT17            = (score < 17);         "Stand
        enable led      = !ena;

truth_table (score -> led)
        @const i = 0;
        @repeat 32 {
    @expr i;     -> @expr {digit} i%10; ; @const i = i+1;}

test_vectors ([ena,score] -> led)
        @const  i = 0;
        @repeat 32 {
   [ L, @expr i; ] -> @expr {digit} i%10; ; @const i = i+1;}

   [ H ,  0] -> HighZ;
   [ H , 15] -> HighZ;
   [ H , 31] -> HighZ;
end led1
```

Listing 6-2. Source File: LED1, Low-Order Display Decoder

LED2, the display decoder for the high-order LED, is described in much the same manner. LED2 differs, however, in that it has two sets of inputs, one from the adder that represents the current score, and the other from the state machine that indicates the current state of the machine. LED2 produces the display signals either for the high-order digit of the score or for the four states, ShowHit, ShowStand, ShowBust and Wait. Listing 6-3 shows the source file for LED2.

The truth table for the LED2 score display is described with directives as in LED1 except that the integer division operator is used rather than the modulus used in LED1. The decode function for the ShowHit, ShowStand, ShowBust and Wait states is described in a separate truth table that decodes just those four states. Two sets of test vectors are also written to test the two different decode functions. The state declarations in the file BJSTATE.TAB are included with @INCLUDE.

```
module  led2   flag '-r2'
title
'Binary to seven segment display decoder for Blackjack machine
Michael Holley & Walter Bright     Data I/O Corp  24 Feb 1984'
"
"           a        binary-to-seven-segment decimal decoder
"          ---
"        f| g |b
"         ---          segment identification
"        e| d |c
"          ---
" digit0  digit1  digit2  digit3  blank    hit     stand    bust
"  ---             ---     ---                      ---
" |   |       |       |       |           |   |   |       |
"                  ---     ---          ---     ---     ---
" |   |       |   |           |           |   |       |   |   |
"  ---             ---     ---                      ---     ---
"
        BJ5A    device  'F82S153';

    S4,S3,S2,S1,S0  pin 1,2,3,4,5;              score  =[S4,S3,S2,S1,S0];
    Q3,Q2,Q1,Q0     pin 6,7,8,9;                Qstate =[Q3,Q2,Q1,Q0];
    a,b,c,d,e,f,g   pin 12,13,14,15,16,17,18;  led    =[a,b,c,d,e,f,g];

        ena                         pin  11;

        ON,OFF  = 1,0;              " for common anode LEDs
        L,H,X,Z = 0,1,.X.,.Z.;
        XXXX    = [.X.,.X.,.X.,.X.];

@include 'bjstate.tab'

"       input                       outputs
"                    a    b    c    d    e    f    g
        digit0  = [OFF, OFF, OFF, OFF, OFF, OFF, OFF];
        digit1  = [OFF,  ON,  ON, OFF, OFF, OFF, OFF];
        digit2  = [ ON,  ON, OFF,  ON,  ON, OFF,  ON];
        digit3  = [ ON,  ON,  ON,  ON, OFF, OFF,  ON];
        blank   = [OFF, OFF, OFF, OFF, OFF, OFF, OFF];
        hit     = [OFF,  ON,  ON, OFF,  ON,  ON,  ON];
        stand   = [ ON, OFF,  ON,  ON, OFF,  ON,  ON];
        bust    = [OFF, OFF,  ON,  ON,  ON,  ON,  ON];

        highZ   macro { [Z, Z, Z, Z, Z, Z, Z]};

truth_table
        ([ena,score] -> led)
        @const i = 0;
        @repeat 32 {
         [ L, @expr i; ] -> @expr {digit} i/10; ; @const i = i+1;}

truth_table
        ([ena,Qstate   ] -> led)
         [ H, Wait     ] -> blank;
         [ H, ShowHit  ] -> hit;
         [ H, ShowStand] -> stand;
         [ H, ShowBust ] -> bust;
@page

test_vectors  'Test digits'
       ([ena,Qstate   ,score] -> led )
        @const i = 0;
        @repeat 27 {
        [ L ,ShowStand,  i  ] -> @expr {digit} i/10; ;
        [ L ,ShowBust ,  i  ] -> @expr {digit} i/10; ;
        [ L ,ShowHit  ,  i  ] -> @expr {digit} i/10; ; @const i = i+1;}

test_vectors  'Test Hit, Stand, Bust'
       ([ena,Qstate   ,score] -> led )
        @const i = 0;
        @repeat 27 {
        [ H ,Wait     , i   ] -> blank;
        [ H ,ShowStand, i   ] -> stand;
        [ H ,ShowHit  , i   ] -> hit;
        [ H ,ShowBust , i   ] -> bust;  @const i=i+1;}

end led2
```

Listing 6-3. Source File: LED2, High-Order Display Decoder

ADD5 is a five-bit adder implemented in an F82S159 IFL that adds the values present at its inputs to the current adder outputs. ADD5 has the added feature that it clears when all the value inputs from CARDSEL are high (value of 31). Listing 6-4 shows the source file for ADD5. The adder cannot be described with the addition operator (+) because too many product terms for the device are produced. Instead, macros provide the intermediate carry terms used to form the complete sum. The clear function is described by a simple equation that uses the internal preset nodes of the IFL. The nodes were declared in the declarations section of the source file. *PA* and *PB* are set high when *V0–V4* are high. Notice that the comparison is done with the set, *Clr*, which was defined as the constant set, ^b11111. For more information on the internal nodes of IFLs and how to use them, see section 5. Test vectors for ADD5 are straightforward and test both the add and clear functions.

```
  module ADD5
title '5-bit ripple adder
Michael Holley & Mary Bailey    Data I/O  Redmond WA   24 Feb 1984'

        BJ2A    device  'F82S159';

" Equations for a n-Bit ripple adder
"       Sn   = An xor Bn xor Cn
"       Cn+1 = AnBn + (An + Bn)Cn
"
        C,L,H,Clr       = .C.,0,1,^b11111;
        Clk,Ena         pin 1,11;
        V4,V3,V2,V1,V0  pin 2,3,4,5,6;
        S4,S3,S2,S1,S0  pin 13,14,15,16,17;
        PA,PB           node 24,22;

" Compute the first two carries with macros
        C0      macro   {0};
        C1      macro   {(V0 & S0 # (V0 # S0) & C0)};

" Compute these carries separately to save product terms
        C2      pin     7;
        C3      pin     8;
        C4      pin     9;

equations
        C2       = V1 & S1 # (V1 # S1) & C1;
        C3       = V2 & S2 # (V2 # S2) & C2;
        C4       = V3 & S3 # (V3 # S3) & C3;

        S0      := V0 $ S0 $ C0;
        S1      := V1 $ S1 $ C1;
        S2      := V2 $ S2 $ C2;
        S3      := V3 $ S3 $ C3;
        S4      := V4 $ S4 $ C4;

equations " for asynchronous clear

        [PB,PA]  = ([V4,V3,V2,V1,V0] == Clr);  " Preset

test_vectors ([Clk,Ena, [V4,V3,V2,V1,V0] ] -> [S4,S3,S2,S1,S0])

        [C,L,Clr] ->  0;        "clear
        [C,L,  7] ->  7;
        [C,L, 10] -> 17;
        [L,L,Clr] ->  0;        "no clock - should clear
        [C,L,  1] ->  1;
        [C,L, 10] -> 11;        "Add 10 to convert Ace to 11
        [C,L,  4] -> 15;
        [C,L,  8] -> 23;        "Subtract 10 by adding 22
        [C,L, 22] -> 13;        "45 overflows to 13
        [C,L,  5] -> 18;
end ADD5
```

Listing 6-4. Source File: ADD5, Five-Bit Ripple Adder

CARDSEL is implemented in a P16HD8, as shown in listing 6–5. The state inputs, *Q0–Q3*, and the card reader inputs, *D0–D3*, are given pin assignments. The outputs pins for *valid*, *is_ace*, and *V0–V4* are declared. The state and card inputs and the outputs are grouped into sets named *Qstate*, *card*, and *value*. The set *card* is declared with five elements, even though the card input has only four bits, so that set operations can be performed in the equations section. (Remember that two-set operations require sets of equal size.) The fifth bit of *card* is defined as a don't care value, so it does not affect the operations. Other constant sets are defined to supply special values to be passed to the adder.

The card selector function is described with Boolean equations. The *is_ace* and *valid* signals require only simple comparison equations. The value outputs are determined through a combination of multiplexing and ANDing functions. When the state machine is in the *Add10*, *Sub10*, and *Clear* states, specific values must be passed to the adder so that it adds, subtracts, or clears as desired. The required value is ANDed with a comparison between *Qstate* (the current state) and the state that requires that value. This occurs in the equation for *value*. Note also that when the machine is in *AddCard* state and a valid card is read, the value is ANDed with the set, [0,1,1,1,1], to ensure that the high-order bit of *value* is 0.

The test vectors are written to check the multiplexing and comparison functions. Two test vectors sections are written, one to test the passing of card reader inputs when the state machine is in the *AddCard* state, the other to check the passing of the fixed values to the adder when the state machine is in *Add10*, *Sub10*, and *Clear* states. The state definitions are included in the module with the @INCLUDE directive.

```
module  Cardsel
title 'Card input selector for Blackjack Machine
Michael Holley      Data I/O Redmond WA     13 Jan 1984'

        BJ1A            device 'P16HD8';

        H,L,X           = 1,0,.X.;


           Q3,Q2,Q1,Q0  pin 5,6,7,8;          Qstate =    [Q3,Q2,Q1,Q0];
        valid,is_ace    pin 17,18;

           D3,D2,D1,D0  pin 1,2,3,4;          card   = [ X,D3,D2,D1,D0];
        V4,V3,V2,V1,V0  pin 12,13,14,15,16;   value = [V4,V3,V2,V1,V0];

        ten             = [0,1,0,1,0];
        minus_ten       = [1,0,1,1,0];
        clr             = [1,1,1,1,1];
        blank_msb       = [0,1,1,1,1];

@include 'bjstate.tab'
equations
        is_ace          = (!card == 1);

        valid           = (!card >= 1) & (!card <=10);

        value = (Qstate==AddCard) & valid & !card & blank_msb
              # (Qstate==Add10)   & ten
              # (Qstate==Sub10)   & minus_ten
              # (Qstate==Clear)   & clr;
@page
test_vectors  'Test card values'
       ([Qstate,card] ->  [value,valid,is_ace])
        [AddCard, !0] ->  [   0,    L,     L];
        [AddCard, !1] ->  [   1,    H,     H];
        [AddCard, !2] ->  [   2,    H,     L];
        [AddCard, !3] ->  [   3,    H,     L];
        [AddCard, !4] ->  [   4,    H,     L];
        [AddCard, !5] ->  [   5,    H,     L];
        [AddCard, !6] ->  [   6,    H,     L];
        [AddCard, !7] ->  [   7,    H,     L];
        [AddCard, !8] ->  [   8,    H,     L];
        [AddCard, !9] ->  [   9,    H,     L];
        [AddCard,!10] ->  [  10,    H,     L];
        [AddCard,!15] ->  [   0,    L,     L];

test_vectors  'Test fixed values'
       ([Qstate,card] ->  [value,valid,is_ace])
        [Add10  , 0]  ->  [  10,    X,     X];
        [Add10  , 5]  ->  [  10,    X,     X];
        [Add10  ,15]  ->  [  10,    X,     X];
        [Sub10  , 0]  ->  [  22,    X,     X];
        [Sub10  , 5]  ->  [  22,    X,     X];
        [Sub10  ,15]  ->  [  22,    X,     X];
        [Clear  , 0]  ->  [  31,    X,     X];
        [Clear  , 5]  ->  [  31,    X,     X];
        [Clear  ,15]  ->  [  31,    X,     X];
end Cardsel
```

Listing 6-5. Source File: CARDSEL, Card Selector

The state machine controller, CONTROL, is described with the ABEL™ STATE_DIAGRAM structure, as shown in listing 6-6. Pin, set and constant declarations are made as usual in the declarations section of the source file. The STATE_DIAGRAM structure allows easy and direct specification of the state machine. For each state, the next state and outputs are defined, thus fully describing the blackjack machine's control lines and state transitions.

The test vectors for the state machine are extensive and test some of the possible state transitions. The test vectors are broken into two parts, one which tests simple transitions based on the total score, the other which tests the state transitions when aces are present. Each of the input values is listed explicitly on the input side of the test vector. On the output side, the expected values for the control line *Ace*, the state set, and the adder clock are listed for each combination of inputs. Note the use of the declared constants.

```
module Control     flag '-r2'
title 'BlackJack state machine controller
Michael Holley  Data I/O Corp.  12 Mar 1984'

        BJ3A    device  'P16R6';

"Inputs
        LT17,LT22       pin 2,3;        "Score less than 17 and 22
        is_Ace          pin 4;          "Card is ace
        CardIn,CardOut  pin 5,6;        "Card present switches
        Restart         pin 7;          "Restart game
        ClkIN           pin 8;          "System clock
        Clk,Ena         pin 1,11;

"Outputs
        AddClk          pin 12;         "Adder clock
        Ace             pin 13;
        Q3,Q2,Q1,Q0     pin 14,15,16,17;
        Qstate          =  [Q3,Q2,Q1,Q0]; "state machine registers

        True,False      = 1,0;
        High,Low        = 1,0;
        H,L,C,X         = 1,0,.C.,.X.;  "test vector charactors

@include 'bjstate.tab'

@page
state_diagram Qstate

        State ShowHit:  !AddClk   = !Low;
                        Ace       := Ace;
                        if (CardIn==Low) then AddCard else ShowHit;

        State AddCard:  !AddClk   = ClkIN;
                        Ace       := Ace;
                        if (is_Ace & !Ace) then Add10  else Wait;

        State Add10:    !AddClk   = ClkIN;
                        Ace       := True;
                        goto       Wait;

        State Wait:     !AddClk   = !Low;
                        Ace       := Ace;
                        if (CardOut==Low) then Test17 else Wait;

        State Test17:   !AddClk   = !Low;
                        Ace       := Ace;
                        if LT17 then ShowHit else Test22;

        State Test22:   !AddClk   = !Low;
                        Ace       := Ace;
                        case      LT22          : ShowStand;
                                 !LT22 & !Ace   : ShowBust;
                                 !LT22 &  Ace   : Sub10;
                        endcase;

        State Sub10:    !AddClk   = ClkIN;
                        Ace       := False;
                        goto       Test17;

        State ShowBust: !AddClk   = !Low;
                        Ace       := Ace;
                        if (Restart==Low) then Clear else ShowBust;

       State ShowStand: !AddClk   = !Low;
                        Ace       := Ace;
                        if (Restart==Low) then Clear else ShowStand;

        State Clear:    !AddClk   = ClkIN;
                        Ace       := False;
                        if Restart then ShowHit else Clear;
```

Listing 6-6. Source File: CONTROL, State Machine Controller

Listing continued on next page.

```
#page
test_vectors 'Assume two cards that total between 16 and 21'
([Ena,Clk,ClkIN,LT17,LT22,is_Ace,Restart,CardIn,CardOut]->[Ace,Qstate,AddClk]
    [L, C,   L,     H,   H,     L,      L,      H,       L  ]->[X,  Clear     ,H];
    [L, C,   L,     H,   H,     L,      L,      H,       L  ]->[L,  Clear     ,H];
    [L, C,   L,     H,   H,     L,      H,      H,       L  ]->[L,  ShowHit   ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       H  ]->[L,  ShowHit   ,L];
    [L, C,   L,     H,   H,     L,      H,      L,       H  ]->[L,  AddCard   ,H];
    [L, C,   L,     H,   H,     L,      H,      L,       H  ]->[L,  Wait      ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       H  ]->[L,  Wait      ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       L  ]->[L,  Test17    ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       L  ]->[L,  ShowHit   ,L];
    [L, C,   L,     H,   H,     L,      H,      L,       H  ]->[L,  ShowHit   ,L];
    [L, C,   L,     H,   H,     L,      H,      L,       H  ]->[L,  AddCard   ,H];
    [L, C,   L,     L,   H,     L,      H,      L,       H  ]->[L,  Wait      ,L];
    [L, C,   L,     L,   H,     L,      H,      H,       H  ]->[L,  Wait      ,L];
    [L, C,   L,     L,   H,     L,      H,      H,       L  ]->[L,  Test17    ,L];
    [L, C,   L,     L,   H,     L,      H,      H,       L  ]->[L,  Test22    ,L];
    [L, C,   L,     L,   H,     L,      H,      H,       L  ]->[L,  ShowStand,L];
    [L, C,   L,     L,   H,     L,      H,      H,       L  ]->[L,  ShowStand,L];
    [L, C,   L,     L,   H,     L,      L,      H,       L  ]->[L,  Clear     ,H];

test_vectors 'Assume 2 Aces and another card that total between 16 and 21'
    [L,Clk,ClkIN,LT17,LT22,is_Ace,Restart,CardIn,CardOut]->[Ace,Qstate,AddClk]
    [L, C,   L,     H,   H,     L,      L,      H,       L  ]->[L,  Clear     ,H];
    [L, C,   L,     H,   H,     L,      H,      H,       L  ]->[L,  ShowHit   ,L];
    [L, C,   L,     H,   H,     H,      H,      H,       H  ]->[L,  ShowHit   ,L];
    [L, C,   L,     H,   H,     H,      H,      L,       H  ]->[L,  AddCard   ,H];
    [L, C,   L,     H,   H,     H,      H,      L,       H  ]->[L,  Add10     ,H];
    [L, C,   L,     H,   H,     H,      H,      L,       H  ]->[H,  Wait      ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       H  ]->[H,  Wait      ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       L  ]->[H,  Test17    ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       L  ]->[H,  ShowHit   ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       H  ]->[H,  ShowHit   ,L];
    [L, C,   L,     H,   H,     L,      H,      L,       H  ]->[H,  AddCard   ,H];
    [L, C,   L,     H,   H,     L,      H,      L,       H  ]->[H,  Wait      ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       H  ]->[H,  Wait      ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       L  ]->[H,  Test17    ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       L  ]->[H,  ShowHit   ,L];
    [L, C,   L,     H,   H,     L,      H,      H,       H  ]->[H,  ShowHit   ,L];
    [L, C,   L,     H,   H,     L,      H,      L,       H  ]->[H,  AddCard   ,H];
    [L, C,   L,     L,   H,     L,      H,      L,       H  ]->[H,  Wait      ,L];
    [L, C,   L,     L,   H,     L,      H,      H,       H  ]->[H,  Wait      ,L];
    [L, C,   L,     L,   H,     L,      H,      H,       L  ]->[H,  Test17    ,L];
    [L, C,   L,     L,   H,     L,      H,      H,       L  ]->[H,  Test22    ,L];
    [L, C,   L,     L,   H,     L,      H,      H,       L  ]->[H,  ShowStand,L];
    [L, C,   L,     L,   H,     L,      H,      H,       L  ]->[H,  ShowStand,L];
    [L, C,   L,     L,   H,     L,      L,      H,       L  ]->[H,  Clear     ,H];
end_Control
```

Listing 6-6. *(Continued)*

6809 SYSTEM CONTROLLER 6.2

This design uses one P16L8 PAL, two P14H4 PALs, and a 74LS393 dual counter to provide the address decoding and multiplexing, the control lines, and the refresh counter for a 6809 microprocessor system. Designs for the multiplexer and address decoder are taken from section 2 and incorporated here into a full system design. The choices made between programmable logic and TTL logic are discussed.

Design Specification

Figure 6–4 shows the system schematic. The 6809 microprocessor provides address lines to three memory types: ROM, memory mapped I/O and dynamic RAM. The type of memory to be accessed is selected by decoding the six high–order bits of the 16–bit address. Table 6–4 shows the correlation between memory types and addresses. The ROM is addressed by *A0–A12*. I/O addressing depends on the devices used. The dynamic RAM is addressed by all 16 bits of the address, *A0–A15*. In addition, control lines and memory refresh must be supplied to the dynamic RAM.

Table 6–4. Address Ranges For a 6809 Microprocessor System

Address Range	Selected Memory
0000–DFFF	Dynamic RAM
E000–E3FF	I/O
E400–EFFF	Dynamic RAM
F000–FFFF	PROM

Address decoding and the creation of the dynamic RAM control lines (RAS and CAS) are provided by a P16L8 PAL programmed to perform these functions. Refresh and addressing of the dynamic RAM is handled with a TTL 74LS393 counter and two identical multiplexers implemented in P14H4s. Operation of the ROM (a 2764 EPROM) and the I/O are not discussed further in any detail, their operation is straightforward and requires only that the programmable logic devices enable the correct memory when it is to be accessed.

The multiplexing and refresh function could have been implemented in many ways incorporating varying degrees of programmable and TTL logic. Table 6–5 shows three of the possible device choices including the one used in this example. Space, power, chip count, and cost must all be taken into account in choosing an approach. Method 2 was chosen because it reduces the chip count, yet is moderate in power consumption.

Table 6–5. Design Approaches for Multiplexing and Refresh

Method	Devices Required	Total Number of Devices	Typical Power	Maximum Power
1	74LS153 (4) 74LS393	5	40 mA	65 mA
2	P14H4 (2) 74LS393	3	125 mA	200 mA
3	P20R4 (2)	2	320 mA	420 mA

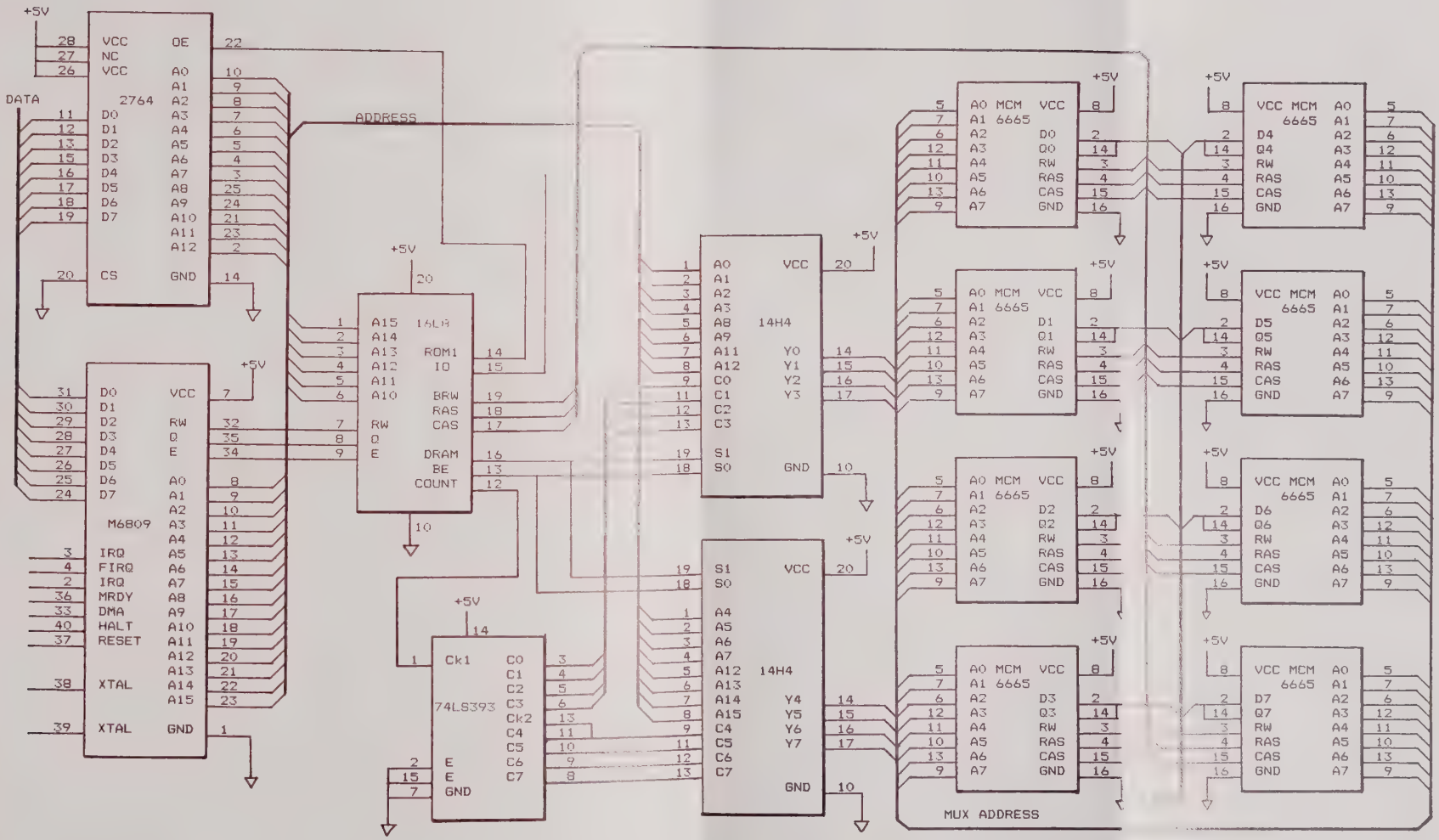


Figure 6-4. System Schematic: 6809 Microprocessor System

The MCM 6665 dynamic RAM used in this design must be completely refreshed every two milliseconds. This is accomplished by cycling through the dynamic RAM addresses with the 74LS393 counter at any time the RAM is not accessed for a read or write. Whenever the microprocessor is accessing ROM or I/O, or performing internal operations, the dynamic RAM is being refreshed. Timing is such that the memory is always refreshed more often than required. (The 6809 provides a 1–MHz clock.)

The column address strobe (CAS) and row address strobe (RAS) must be provided after the address and data have stabilized. Because the 6665 is provided in a 16–pin package, only eight address pins are available to access 2^{16} (65,536) addresses. This requires that the low–order and high–order address bits be multiplexed in, with *A0–A7* available during the first half of the read/write cycle and *A8–A15* available during the second half of the cycle. Figure 6–5 shows a timing diagram for the dynamic RAM addressing and refresh functions as the RAM is first read, then written, then refreshed.

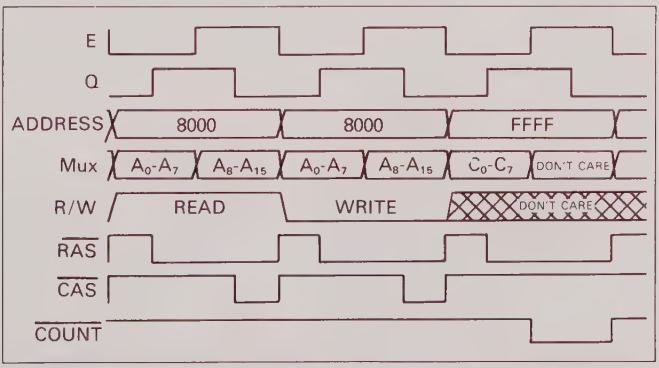


Figure 6–5. Timing Diagram: 6809 Dynamic RAM Access

The 1-MHz clocks, *E* and Q, provided by the microprocessor are shown at the top of the figure. *CAS* and *RAS* are generated from *E* and Q as shown. When the address is in a valid range, *DRAM* goes low and the dynamic RAM is accessed for a read or write. During the first half of a read or write cycle, the multiplexer, *MUX*, must route the low-order address bits to the RAM; during the second half of the cycle, the high-order address bits are provided. When the address is outside the dynamic RAM range (as determined by the six high-order bits), *MUX* must provide the counter output as a RAM address for refresh.

Design Method

The design is described with ABEL™ in two separate modules, REF6809 and MUX12T4. REF6809 is a complete design for the address decoder and control line generator implemented with a PAL P16L8. MUX12T4 is a simple 12-line to 4-line multiplexer implemented in a P14H4 and is used to provide the necessary address multiplexing. The design for MUX12T4 is identical to that presented in section 2.2 and is not discussed here. These separate modules can be processed together or separately by the language processor. Two programmer load files will be generated, one for the decoder and another for the multiplexers. These load files are then used to program the required parts (one decoder and two multiplexers).

REF6809 is designed using three separate sets of Boolean equations, as shown in the source file, listing 6-7. The test vectors are also written in three sections, one for each set of equations. The pin and device declarations are made and a set named *Address* is defined to simplify notation in the test vectors.

The first equations section describes the address decoding function. Simple relational expressions are used to select the appropriate type of memory as determined by the six high-order address lines. The test vectors following the equations test that decode function.

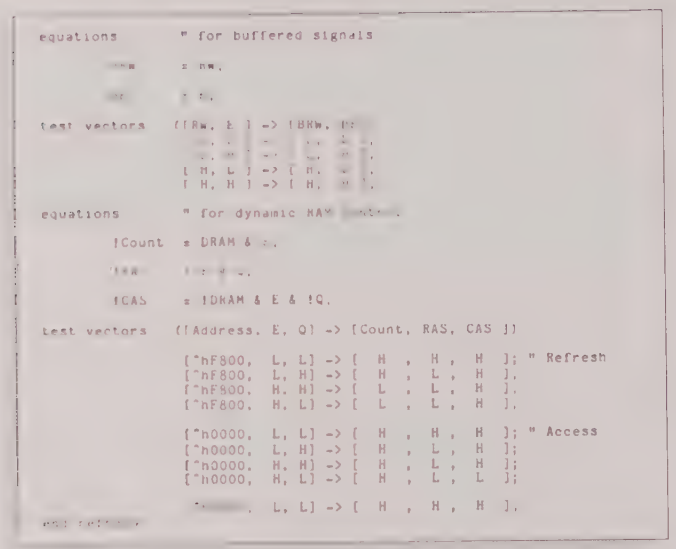
The second equations section buffers the *E* clock and *RW* provided by the 6809 through simple assignments, and another test vectors section is written to test that function.

The third equations section specifies the control line generation. *Count* is a clock line provided to the 74LS393 counter when the dynamic RAM is not being read or written. *RAS* and *CAS* are generated from *E*, *Q*, and *DRAM*. Note that *DRAM* can be used as an input to the *Count* and *CAS* equations even though it is declared as an output of the design (see the pin declaration) because it is fed back internally. The third test vectors section tests the control line generation.

```
module ref6809
title '6809 memory decode and DRAM control
Gerry Simmons     Data I/O Corp Redmond WA   24 Feb 1984'

                          U09      device  'P16L8';
        A15,A14,A13,A12,A11,A10 pin 1,2,3,4,5,6;
        PROM,IO,DRAM            pin 14,15,16;
        RW,E,Q                  pin 7,8,9;
        Count,BE,CAS,RAS,BRW    pin 12,13,17,18,19;

        H,L,X                   = 1,0,.X.;

" The SET notation is used to group the 6 address bits into a word
        Address = [A15,A14,A13,A12, A11,A10,X,X, X,X,X,X, X,X,X,X];

equations       " for address decode

        !PROM   = (Address >= ^hF000);

        !IO     = (Address >= ^hE000) & (Address <= ^hE3FF);

        !DRAM   = (Address <= ^hDFFF)
                # (Address >= ^hE400) & (Address <= ^hEFFF);

test_vectors (Address -> [PROM,IO,DRAM])
                ^h0000  -> [  H,  H,  L ];
                ^h4000  -> [  H,  H,  L ];
                ^h8000  -> [  H,  H,  L ];
                ^hC000  -> [  H,  H,  L ];
                ^hE000  -> [  H,  L,  H ];
                ^hE400  -> [  H,  H,  L ];
                ^hEE00  -> [  H,  H,  L ];
                ^hF000  -> [  L,  H,  H ];
                ^hF800  -> [  L,  H,  H ];

@page
```

Listing 6–7. Source File: 6809 Microprocessor Memory Controller

Listing continued on next page.



```
equations       " for buffered signals

                = RW.

                = E.

test vectors    ([RW, E ] -> [BRW, E ]
                 [ L, L ] -> [ L,  L ].
                 [ L, H ] -> [ L,  H ].
                 [ H, L ] -> [ H,  L ].
                 [ H, H ] -> [ H,  H ].

equations       " for dynamic RAM

        !Count  = DRAM & Q.

        !RAS    = !DRAM & E & !Q.

        !CAS    = !DRAM & E & !Q.

test vectors    ([Address, E, Q] -> [Count, RAS, CAS ])

                [^hF800,  L, L] -> [  H  ,  H ,  H  ]; " Refresh
                [^hF800,  L, H] -> [  H  ,  L ,  H  ].
                [^hF800,  H, H] -> [  L  ,  L ,  H  ].
                [^hF800,  H, L] -> [  L  ,  L ,  H  ].

                [^h0000,  L, L] -> [  H  ,  H ,  H  ]; " Access
                [^h0000,  L, H] -> [  H  ,  L ,  H  ];
                [^h0000,  H, H] -> [  H  ,  L ,  H  ];
                [^h0000,  H, L] -> [  H  ,  L ,  L  ];

                [^hFFFF,  L, L] -> [  H  ,  H ,  H  ].
```

Listing 6–7. *(Continued)*

INDEX AND REFERENCE

DATA I/O

INDEX

The following index covers all three manuals contained in this binder. Each index entry is keyed to indicate the manual in which it is located, as follows:

UG = User's Guide
LR = Language Reference
AG = Applications Guide

A
-A, UG:4-9
Abellib.dev, UG:4-4, UG:6-4
Active low output enable, AG:5-3
Actual arguments, LR:2-27, LR:3-14
Adder, AG:6-5
Address decoder, AG:2-2
Adjacent states, AG:3-6
Alphabetic characters, LR:2-9
ALTERNATE directive, LR:4-2
Alternate operator set, LR:4-2
Alternate operators, LR:4-2
Arguments, LR:2-27, LR:3-4, AG:3-16
  actual, LR:2-27, LR:3-14
  dummy, LR:2-27, LR:3-14
  pass to source, UG:4-9
  substitution, LR:2-27
Arithmetic operators, LR:2-12
ASCII
  characters, LR:2-2
  table, LR:B-1
Assignment operators, LR:2-15, LR:2-18
Assignments
  clocked (combinational), LR:2-25
  unclocked (registered), LR:2-25
Asynchronous circuits, AG:4-24
Asynchronous feedback circuit, UG:4-36
Attributes, LR:3-17
  feedback, LR:3-18
  latch, LR:3-19
  pin, LR:3-9

B
-B, UG:4-25
Back-quotes, LR:2-6

Barrel shifter, AG:2-22
Base numbering, change, LR:4-12
Batch file, UG:4-1, UG:4-3, UG:4-6
Bidirectional buffer, AG:2-36
Bit equality, AG:2-33
Bit values, AG:3-7
Blackjack machine, AG:6-2
Blanks, LR:2-28
Block repeat, LR:4-9
Blocks, LR:2-26
Blocks, nesting of, LR:2-26
Boolean algebra, LR:3-21
Braces, LR:2-26, AG:3-15
Brackets, UG:1-15
Breakpoints, UG:4-25, AG:4-2
Buried registers, AG:5-31

C
-C, UG:4-20
CASE, LR:3-32
Characters
  alphabetic, LR:2-9
  form feed, LR:2-1
  line feed, LR:2-1
  vertical tab, LR:2-1
Checksum parameter, UG:4-20
Chip diagram, UG:4-40
Circuit redundancy, AG:3-12
Circumflex, LR:2-9
Clear function, AG:6-14
Cleared state, AG:3-5
Cleared-register state, AG:3-5
Clock inputs, AG:4-14
Clock pulse, UG:4-32
Combinational outputs, LR:3-23
Combining equations, AG:3-8, AG:5-25
Command line, UG:3-14, AG:3-14
Command line errors, UG:A-2
Comments, LR:2-7
Comparator, AG:2-30
Complement
  array, AG:5-31
  array feedback, AG:5-31
  operators, LR:2-18

Complex structures, LR:4-1
Configurable output enable, AG:5-4
CONST (constant) directive, LR:4-3
Constant assignments, AG:2-13
Constant declaration, UG:3-10, LR:3-12, AG:2-19
Constant values, LR:3-12, AG:3-2
Constant values, special, LR:2-10
Constants, LR:2-10, LR:3-7, LR:3-13, AG:2-18, AG:2-26, AG:4-9
Constants, collection of, LR:2-20
Conversion program, UG:6-2
Conversion utility, UG:6-1
Created files, UG:3-16

D
-D, UG:4-20
D-type flip-flops, AG:5-18
DASH-ABEL, UG:1-1, UG:1-4
Data translation format, UG:4-21
Decade counter, AG:5-31
Decimal counter, AG:2-17
Decimal values, LR:2-9
Declaration
  constant, UG:3-10, LR:3-12
  device, UG:3-9, LR:3-7, LR:3-11
  macro, LR:3-14
  node, UG:3-9, LR:3-10, AG:5-20
  pin, UG:3-9, LR:3-8
  section, LR:3-7, AG:2-6
  statements, LR:3-7, AG:5-13
Declarations, LR:3-7
  node, AG:5-20
Demultiplexer, AG:2-8
Design
  analysis, AG:4-2
  documentation, UG:4-39
  documentation file, UG:4-5
  examples, UG:1-12, AG:1-2
  with feedback, UG:4-32
Device
  declaration, UG:3-9, LR:3-7, LR:3-11
  file errors, UG:A-14
  identifier, LR:3-23
  library, UG:4-4, UG:6-4
  nodes, AG:5-11

specification files, UG:4-10
types, AG:3-14
DEVLIB, UG:6-4
Diagrams syntax, LR:A-1
Directive
alternate, LR:4-2
constant, LR:4-3
exit, LR:4-4
EXPR (Expression), LR:4-4
IF, LR:4-4
IFB (If Blank), LR:4-5
IFDEF (If Defined), LR:4-5
IFIDEN (If Identical), LR:4-6
IFNB (If Not Blank), LR:4-6
IFNDEF (If Not Defined), LR:4-7
IFNIDEN (If Not Identical), LR:4-7
INCLUDE, LR:4-8
IRP (Indefinte Repeat), LR:4-9
IRPC (Indefinite Repeat Character), LR:4-10
MESSAGE, LR:4-11
PAGE, LR:4-11
RADIX, LR:4-12
REPEAT, LR:4-13
STANDARD, LR:4-13
Directives, LR:2-26, AG:3-14, AG:4-6
list, UG:4-9
usage, LR:4-1
Display decoder, AG:2-26
Document, UG:4-39, UG:4-42
Document output file, UG:4-42
Documentation, UG:4-39
Documentation file, UG:3-16
Don't care
flag, AG:4-11
in simulation, AG:4-11
in test vectors, UG:4-26
inputs, AG:4-11
outputs, AG:4-11
Don't-care values, AG:2-37
Downloading, UG:5-1
Downloading programmer load file, UG:3-17
Dummy arguments, LR:2-27, LR:3-14, AG:3-14, AG:3-15, AG:4-7

E
-E, UG:4-9
Elements, LR:2-18
Ellipsis, UG:1-15, LR:1-3
ENABLE keyword, LR:2-18
Enable outputs, AG:2-9
Enables, AG:2-9
Equation polarity reduction speed, AG:3-18
Equations, UG:4-40, LR:2-18, LR:3-21, AG:2-13, AG:2-28
  combination of, AG:3-8
  combining, AG:5-25
  ORed, AG:5-25
  ORing of, AG:3-9
Equations section, AG:2-13
Equations statement, UG:3-11, LR:3-21
Error messages, UG:A-2
  command line, UG:A-2
  device file errors, UG:A-14
  fatal errors, UG:A-3
  IFLDOC errors, UG:A-17
  intermediate file errors, UG:A-5
  internal errors, UG:A-14
  logical errors, UG:A-7
  parse listing, UG:A-11
  proprocessor errors, UG:A-10
  simulation errors, UG:A-15
  syntax errors, UG:A-11
  TOABEL errors, UG:A-16
Example
  adder, AG:6-5
  barrel shifter, AG:2-22
  bidirectional buffer, AG:2-36
  blackjack machine, AG:6-2
  comparator, AG:2-30
  decimal counter, AG:2-17
  multiplexer, AG:2-5
  seven-segment decoder, AG:6-5
  state machine, AG:6-2
  state machine controller, AG:6-18
  synchronous counter, AG:2-11
  system controller, AG:6-21
Examples, general, UG:1-12, AG:1-2, AG:2-1
Examples disk, UG:1-12
Exclusive-NOR, AG:2-33

Exclusive-OR operators, AG:2-31
EXIT directive, LR:4-4
Expanded output, UG:4-9
Expanded source code, UG 4-9
EXPR (Expression) directive, LR:4-4
Expressions, LR:2-16, LR:2-18

F
-F, UG:4-41
Fatal errors, UG:A-3
Feedback, UG:4-32
  attributes, LR:3-18
  circuits, UG:4-32
  from OR-gate, LR:3-18
  from output pin, LR:3-18
  from register, LR:3-18
  paths, LR:3-18
  registers, AG:5-31
File
  batch, UG:4-1
  design documentation, UG:4-5
  documentation, UG:3-16
  intermediate, UG:3-16, UG:4-7
  parse listing, UG:4-11
  parser listing, UG:4-5
  programmer load, UG:3-16, UG:4-5
  simulate output, UG:4-28
  simulation output, UG:3-16, UG:4-5
Files
  created, UG:3-16
  device specification, UG:4-10
  including external, LR:4-8
  output, UG:4-5
  temporary, UG:4-5
Flag statement, UG:3-8, LR:3-4, LR:3-5, AG:2-26
Flip-flop mode control fuses, AG:5-22
Flip-flops
  configuration of, AG:5-17
  D-type, AG:5-18
  input notation, AG:5-26
  J-K type, AG:5-18
  mode control, AG:5-22
  specifying inputs, AG:5-26
Form feed character, LR:2-1

Format load file, UG:4-20
Fuse map, UG:4-18, UG:4-40
Fuses, LR:3-35
  inaccessible, LR:3-10
  multiple numbers, LR:3-35
  numbers, LR:3-35
  states, LR:3-35
  unused, UG:4-21
Fuses section, LR:3-35

G
Gate outputs, UG:4-30
GOTO, LR:3-33

H
-H, UG:4-10
Hex
  notation, AG:2-4
  values, LR:2-9
High-impedance, AG:2-37
High-impedence
  in test vectors, UG:4-26
  outputs, AG:2-20

I
-I, UG:4-8
Identifiers, LR:2-1, LR:2-3, AG:3-2
  choosing, LR:2-5
  reserved, LR:2-4
  mixed-case, LR:2-5
  signal, LR:3-38
  with operators, LR:2-16
IF directive, LR:4-4
IF-THEN-ELSE, LR:3-30
  chained, LR:3-31
  mutually-exclusive, LR:3-31
IFB (If Blank) directive, LR:4-5
IFDEF (If Defined) directive, LR:4-5
IFIDEN (If Identical) directive, LR:4-6
IFLDOC, UG:6-2
IFLDOC errors, UG:A-17
IFNB (If Not Blank) directive, LR:4-6
IFNDEF (If Not Defined) directive, LR:4-7
IFNIDEN (If Not Identical) directive, LR:4-7

Illegal states, AG:4-20
INCLUDE directive, LR:4-8
Included files, UG:4-10
Input file UG 4-8
Intermediate file, UG:3-16, UG:4-7
Intermediate file errors, UG:A-5
Internal errors, UG:A-14
Inverted outputs, AG:2-24
IRP (Indefinite Repeat) directive, LR:4-9
IRPC (Indefinite Repeat Character) directive, LR:4-10
Istype examples, LR:3-20
Istype statement, LR:3-17, AG:5-6
Italics, UG:1-15, LR:1-3

J
-J, UG:4-20
J-K flip-flops, AG:5-18
JEDEC
  specification, UG:B-1
  file, UG:6-2

K
-K, UG:4-21
Keywords, LR:2-1

L
-L, UG:4-9
Language processor, UG:3-14
Latch attribute, LR:3-19
Less than, AG:2-33
Library, UG:6-4
Library manager, UG:6-4
License, UG:1-18
License agreement/warranty, UG:1-18
Line endings, LR:2-1
Linefeed character, LR:2-1
List equations, UG:4-41
List fuse map, UG:4-41
Listing File, UG:4-9
Logic devices, advanced, AG:5-1
Logic reduction, UG:1-6
Logic reduction times, AG:3-18
Logic redundancy, AG:3-12
Logic values, true-false, LR:2-14
Logical errors, UG:A-7

Logical operators, LR:2-11
Long logic reduction times, AG:3-18
Lower case, UG:1-15, LR:1-3

M
Macro cell
  feedback point, AG:5-8
  polarity, AG:5-6
  type, AG:5-10
Macro cells, AG:5-10, AG:5-17
Macros, LR:2-26, LR:3-7, LR:3-14, AG:2-34, AG:4-6
  declaration, LR:3-14
  expansion, LR:3-14
MESSAGE directive, LR:4-11
Module header, LR:3-6
Module statement, UG:3-7, LR:3-3
Modules, LR:3-3, AG:6-9
Multiplexer, AG:2-5

N
-N, UG:4-25
Negative logic, AG:2-24
Node
  declaration, LR:3-10
  declarations, UG:3-9, AG:5-20
  numbers, AG:5-13, AG:5-26
Nodes, UG:C-1, AG:5-11
  attributes, LR:3-10
  internal, LR:3-10
  programmable, LR:3-10
  shorthand notation, LR:3-11, AG:5-15
Not equal to, AG:2-33
Notation, UG:1-15
Notation conventions, UG:1-15, AG:1-5
Numbers, LR:2-1, LR:2-8
Numeric values, LR:2-8, LR:2-9

O
-O, UG:4-8
Operators, AG:2-31
  assignment, LR:2-15, LR:2-18
  complement, LR:2-18
  exclusive-OR, AG:2-31
  set, LR:C-1

standard, LR:4-13
with identifiers, LR:2-16
Operators and priorities, LR:2-17
ORed equations, AG 5-25
ORing of equations, AG:3-9
Output, high-impedence, AG:4-17
Output control, AG:2-11, AG:2-22, AG:2-24
Output enable, AG:2-14
active low, AG:5-3
configurable, AG:5-4
pin controlled, AG:5-2
term controlled, AG:5-3
Output enable input, AG:5-20
Output file, UG:4-5, UG:4-8
Output macro cell, AG:5-6
Outputs
combinational, LR:3-23
inverted, AG:2-8
macro cell, AG:5-6
registered, UG:4-23, LR:3-23

P
-P, UG:4-9
PAGE directive, LR:4-11
PALASM, UG:6-1
Parse, UG:4-7, UG:A-11
Parse listing, UG:3-16
Parse listing file, UG:4-5, UG:4-11, UG:A-11
Pass arguments to source, UG:4-9
Path
device files, UG:4-10
included files, UG:4-10
programmer load file, UG:4-20
Pin attributes, LR:3-9
Pin controlled output enable, AG:5-2
Pin declaration, UG:3-9, LR:3-8
Pin identifiers, UG:4-30
Pin table, UG:4-30
Power-on state, AG:3-4
Preload, AG:4-13, AG:4-17
supervoltage, AG:4-19
TTL, AG:4-17
with test vectors, AG:4-20
Preprocessor errors, UG:A-10

Preset, AG:4-13, AG:4-14, AG:4-22
  controlled by product term, AG:4-22
PRESTO, UG:4-17
Processing parameters, LR:3-5
Product Support, UG:1-17
Product term, AG:2-23
  too many, AG:3-6
Programmer load file, UG:1-4, UG:1-6, UG:3-16, UG:4-5, UG:5-5
PROMlink, UG:1-1, UG:3-17, UG:5-1
Propagation delays, AG:3-12

Q
-Q, UG:4-41
Quotation marks, UG:1-15, LR:1-3, LR:2-6, LR:2-7

R
RADIX directive, LR:4-12
Reduce, UG:4-15
Reduced equations, UG:4-15
Reduction levels, UG:4-15, UG:4-17
Reduction speed, slow, AG:3-18
Redundancy
  circuit, AG:3-12
  logic, AG:3-12
Refresh function, AG:6-22
Register outputs, UG:4-30
Register type, AG:5-10
Registered outputs, UG:4-23, LR:3-23
Registers
  buried, AG:5-31
  feedback, AG:5-31
Relational expression, AG:2-6
Relational operators, LR:2-13
REPEAT directive, LR:4-13
Reserved identifiers, LR:2-4
Reset, controlled by product term, AG:4-22
Ripple adder, AG:6-5

S
Set
  assignment, LR:2-24
  breakpoints, UG:4-25
  comparison, LR:2-24
  equality, AG:2-33

operations, LR:2-21
restrictions, LR:2-25
Sets, LR:2-20, AG:2-7, AG:2-9, AG:2-13
Seven-segment decoder, AG:6-5
Shift operators, LR:2-12
Shifter/counter, AG:5-17
Shorthand notation, AG:5-26
for nodes, AG:5-15
Signal identifiers, LR:3-38
Signals, collection of, LR:2-20
Simulate, UG:4-22, LR:3-36, AG:4-1, AG:5-29
Simulate output file, UG:4-28
Simulation, UG:1-6, AG:4-1, AG:4-11
Simulation errors, UG:A-15
Simulation output, UG:4-28
Simulation output file, UG:3-16, UG:4-5
Software update, UG:1-17
Source file, UG:1-4
elements of, UG:3-2
processing of, UG:3-13
writing, UG:3-6
Spaces, LR:2-28
imbedded, LR:2-1
Special constant values, LR:2-10
Special constants, AG:5-20
Specific states, AG:3-7
Specify device type, UG:4-25
Square brackets, LR:1-3
STANDARD directive, LR:4-13
Standared operators, LR:4-13
State diagram, LR:3-26, AG:2-19
State machine, LR:3-32, AG:6-2
State machine controller, AG:6-18
State machines, debugging, AG:4-4
State transitions, AG:3-2
Statement
CASE, LR:3-32
chained IF-THEN-ELSE, LR:3-31
declaration, LR:3-7, AG:5-13
equations, UG:3-11
flag, UG:3-8, LR:3-4, LR:3-5, AG:2-26
GOTO, LR:3-33
IF-THEN-ELSE, LR:3-30
module, UG:3-7, LR:3-3

title, UG:3-8, LR:3-4, LR:3-6
WITH-ENDWITH, LR:3-34
Strings and delimiters, LR:2-6
Supervoltage, AG:4-19
Supervoltage preload, AG:4-19
Synchronous counter, AG:2-11
Synchronous feedback circuits, UG:4-32
Syntax errors, UG:A-11
Syntax rules, LR:2-1
System controller, AG:6-21

T
-T, UG:4-24
Temporary files, UG:4-5
Term controlled output enable, AG:5-3
Term minimization, UG:4-17
Test vectors, UG:3-12, UG:4-26, UG:4-28, UG:4-40, LR:3-36, AG:2-4, AG:2-7, AG:2-9, AG:2-14, AG:2-20, AG:2-28, AG:4-2, AG:5-29
header, LR:3-37
multiple tables, AG:4-5
table format, LR:3-37
Title statement, UG:3-8, LR:3-4, LR:3-6
TOABEL errors, UG:A-16
Trace level, UG:4-24
Trace level waveform, UG:4-27
Trace levels, AG:4-2, AG:4-12
Transfor, UG:4-13
Transition equations, AG:5-31
Transitions, AG:3-2, AG:6-7
Tri-state output, LR:2-18
Truth table, LR:3-22, AG:2-28, AG:6-12
format, LR:3-24
header, LR:3-22
long, LR:4-13
TTL preload, AG:4-17

U
Underscores, LR:2-3
Unused fuses, UG:4-21
Upper case, UG:1-15, LR:1-3

V
Variable device type, AG:3-15
Variable values, AG:3-14
Vertical tab character, LR:2-1

W
-W, UG:4-27
Waveform, UG:4-27
WITH-ENDWITH, LR:3-34

X
X, UG:4-26

Y
-Y, UG:4-10

Z
-Z, UG:4-26

GLOSSARY

Asynchronous: An asynchronous circuit is one in which the outputs change at any time, depending on the order in which the inputs change. (See *synchronous*.)

Batch file: A file containing operating system commands to perform a series of operations. Same as *command file*.

Combinational: A combinational circuit is a collection of logic gates with a set of inputs and outputs. The outputs are a direct function of the inputs, and no storage elements are present. Also called combinatorial. (See *sequential*.)

Command file: A file containing operating system commands to perform a series of operations. Same as batch file.

Command line: A line of text entered by the user to start a software program.

DOS: Disk Operating System. Software that controls operation of the components of a computer system and communication among the components.

Directive: An instruction to the language processor that directs it to perform a specific task, such as including text in a source file.

Functional testing: Testing of a programmed logic device to see that it operates as required.

Fuse map: A representation of the fuses in a programmable logic device that shows which fuses are blown (disconnected) or left intact (connected).

Identifier: A name or label used in a source file to reference pins, nodes, sets, constants, keywords, or output signals.

Intermediate file: A file created by a language processor program used to pass data to the next program in the processing. Each program of the language processor creates its own intermediate file.

JEDEC (Joint Electronic Device Engineering Council): Used in this documentation to signify a standard data format for transfer of programming data to logic programmers. Known formally as the JEDEC standard JC-42.1-81-62 Data Transfer Format for Programmable Logic.

Keyword: An identifier that is part of the ABEL™ language.

Language processor: A collection of six programs that process source files to create programmer load files, perform simulation, perform logic reduction, and create design documentation.

Logic description: Boolean equations, truth tables, or state diagrams that describe one or more logic functions.

Logic reduction: The process of reducing user supplied logic descriptions to a near-minimal form so that the number of product terms required to implement a function is reduced.

Macro: A user-defined function that can be used to include text in a source file.

Module: A complete ABEL™ unit that contains everything necessary to describe one or more complete logic functions. One or more modules make up a *source file*.

Nodes: Pseudo-pins defined by Data I/O that have internally accessible signals that do not appear as output pins of the device. Nodes allow the user to write equations using these internal signals.

PRESTO: A logic reduction algorithm developed by Antonin Svoboda. Described in the paper, "A State Machine Synthesizer - SMS," by Douglas W. Brown (**Proceedings of the 18th Design Automation Conference**, IEEE, June 1981, pp. 301-305.)

Parameters: Values passed to a program that direct its operation or indicate what is to be processed by that program.

Parse: To process a source file, checking for correct language syntax, developing tables of user-defined identifiers, and implementing the function of language keywords.

Programmer load file: A file produced by the ABEL™ language processor that contains the fuse states and test vectors necessary to program and test programmable logic devices.

Registered: A registered output is one that is the output of a clocked storage device.

Section: As used in "test vectors section", "equations section", "truth table section", and "state diagram section", indicates one complete truth table, state diagram, test vectors table or group of Boolean equations, associated with a particular logic device.

Sequential: A sequential circuit contains logic gates and storage elements that retain values resulting from earlier input values. (See *combinational*.)

Sets: Groups of constants, input signals, output signals, or nodes that are operated on as a whole according to Boolean algebra rules.

Simulation: The testing of the function of a logic design before a logic device is programmed. Simulation applies user–supplied inputs to the logic function and compares the outputs obtained with predicted outputs.

Source file: Input to the language processor. A file containing one or more complete *modules* that describe logic designs.

State diagram: An ABEL™ construct that allows direct specification of state machine designs in a form that describes next states and outputs as a function of inputs and the current state. The term *state diagram* is also used to describe a pictorial representation of the state transitions of a state machine.

State machine: A logic design containing a state register whose bit value represents the current state, and that undergoes transitions to one state from another based on the current state and inputs.

Synchronous: A synchronous circuit is one in which the outputs change value only upon application of a clock pulse regardless of when the inputs changed.

Syntax The order, structure, and rules of a language.

Test vectors An ABEL™ construct used to specify required or expected outputs for a given set of inputs. Test vectors are used for logic simulation.

Truth table An ABEL™ construct that describes the outputs of a logic design as functions of the inputs in a tabular form.

Twos complement A form of binary complementation in which all the bits of a binary number are complemented (inverted) and then 1 is added to the number. Adding the twos complement of a first number to a second number is the same as subtracting the first from the second.

Software Problem Report

Problem Description: Please describe how to reproduce the problem and include the source file and output files (on disk or tape if possible).

(Use reverse side if more room is needed.)

Name _____

Company _____

Street _____

City _____ State _____ Zip _____

Phone _____ Date _____

Data I/O Software Product _____

Version _____ Serial Number _____

Other Software Used _____

Computer Manufacturer _____ Model _____

Operating System _____ Version _____

Peripherals _____

Programmer Model _____ Revision _____

Mail to: Data I/O Corporation
Customer Support Center
10525 Willows Road N.E./P.O. Box 97046
Redmond, WA 98073-9746
UNIX UUCP uw-beaver!tektone!dataio!bugs

U.S. Sales Offices and Representatives
Direct Sales and Serivce Offices

Data I/O
10525 Willows Road N.E.
P.O. Box 97046
Redmond, WA 98073-9746
(206) 881-6444
Telex Dom. 15-2167,
Int'l 4740166 DIO UI
Fax (206) 882-1043

NORTHERN CALIFORNIA

Data I/O
1700 Wyatt Drive
Suite 1
Santa Clara, CA 95054
(408) 727-0641 *(Sales)*
(408) 727-0659 *(Service)*
Telex 556440

SOUTHERN CALIFORNIA

Data I/O
3505 Cadillac Avenue
Suite L-1
Costa Mesa, CA 92626
(714) 662-1182 *(Sales)*
(714) 662-2498 *(Service)*
Telex 755870

EASTERN REGION

Data I/O
Birch Pond Business Center
22 Cotton Road
Nashua, NH 03063
(603) 889-8511 *(Sales)*
(603) 889-8513 *(Service)*
Telex 943431

CENTRAL REGION

Data I/O
1810 N. Glenville Drive
Suite 108
Richardson, TX 75081
(214) 235-0044
Telex 792474
TWX 910-997-1767

Representatives

ALABAMA

Pen Tech Associates
Huntsville, AL
(205) 881-9298

ALASKA

Northwest T & M
Redmond, WA
(206) 881-8857

ARIZONA

Zeus Electronics, Inc.
Phoenix, AZ
(602) 263-6022
1-800-528-4512
Telex 910-951-1362

ARKANSAS

Testech, Inc.
Richardson, TX
(214) 644-5010

CALIFORNIA, NORTHERN

Data I/O
Santa Clara, CA
(408) 727-0641

CALIFORNIA, SOUTHERN

Data I/O
Costa Mesa, CA
(714) 662-1182
Telex 755870

COLORADO

Zeus Electronics, Inc.
Denver, CO
(303) 321-4246
1-800-521-4512

Addresses on this list subject
to change without notice.

CONNECTICUT
Data I/O
Nashua, NH
(603) 889-8511
Telex 943431

DELAWARE
Sci-Rep, Inc
Baltimore Office
(301) 321-1411

FLORIDA
Pen Tech Associates
Deerfield Beach, FL
(305) 421-4989

Casselberry, FL
(305) 678-6809

FLORIDA, NORTHERN
Pen Tech Associates
Huntsville, AL
(205) 881-9298

GEORGIA
Pen Tech Associates
Marietta, GA
(404) 424-1931

HAWAII
Northwest Test
and Measurement
Redmond, WA
(206) 881-8857

IDAHO
Zeus Electronics, Inc.
Salt Lake City, UT
(801) 534-0500
1-801-534-0503

ILLINOIS, NORTHERN
Torkelson Associates
Deerfield, IL
(312) 945-8700
Twx 910-992-1438

ILLINOIS, SOUTHERN
Palatine Eng. & Sales
Hazelwood, MO
(314) 839-0800

INDIANA
Torkelson Associates
Indianapolis, IN
(317) 244-7867
Telex 810-341-3141

IOWA
Torkelson Associates
Deerfield, IL
(312) 945-8700
Twx 910-992-1438

KANSAS
Palatine Eng. & Sales
Mission, KS
(913) 831-0555

KENTUCKY, WESTERN
Torkelson Associates
Indianapolis, IN
(317) 244-7867
Telex 810-341-3141

KENTUCKY, EASTERN
Electro Sales Associates
Allison Park, PA
(412) 487-3801

LOUISIANA
Testech, Inc.
Houston, TX
(713) 956-0837

MAINE
Data I/O
Nashua, NH
(603) 889-8511
Telex 943431

MARYLAND
Sci-Rep, Inc.
Baltimore Office
(301) 321-1411

MASSACHUSETTS
Data I/O
Nashua, NH
(603) 889-8511
Telex 943431

MICHIGAN
Electro Sales Associates
Livonia, MI
(313) 474-7320

Portage, MI
(616) 323-2416

MINNESOTA
Torkelson Associates
Minneapolis, MN
(612) 835-2414
Telex 910-576-2740

MISSISSIPPI
Pen Tech Associates
Huntsville, AL
(205) 881-9298

MISSOURI
Palatine Eng. & Sales
Hazelwood, MO
(314) 839-0800

MONTANA
Zeus Electronics, Inc.
Salt Lake City, UT
(801) 534-0500
1-800-534-0503

NEBRASKA
Palatine Eng. & Sales
Hazelwood, MO
(314) 839-0800

NEVADA
Zeus Electronics, Inc.
Phoenix, AZ
(602) 263-6022
1-800-528-4512
Telex 910-951-1362

NEW HAMPSHIRE
Data I/O
Nashua, NH
(603) 889-8511
Telex 943431

NEW JERSEY, NORTHERN
Data I/O
1-800-858-5803

NEW JERSEY, SOUTH
Sci-Rep, Inc.
Pennsauken, NJ
(609) 662-5222
Telex 710-892-1297 SCI-REP PESN

NEW MEXICO
Zeus Electronics, Inc.
Albuquerque, NM
(505) 842-6633
1-800-528-4512

NEW YORK, METRO
Data I/O
1-800-858-5803

NEW YORK, UPSTATE
DB Associates, Inc.
Fayetteville, NY
(315) 446-0220

NORTH CAROLINA
Pen Tech Associates
Greensboro, NC
(919) 852-6000

NORTH DAKOTA
Torkelson Associates
Minneapolis, MN
(612) 835-2414
Telex 910-576-2740

OHIO
Electro Sales Associates
Dayton, OH
(513) 426-5551

Cleveland, OH
(216) 261-5440

OKLAHOMA
Testech, Inc.
Richardson, TX
(214) 644-5010

Tulsa, OK
(918) 665-7788

OREGON
Northwest T & M
Beaverton, OR
(503) 646-9966
1-800-247-5700
Telex 910-467-8775

PENNSYLVANIA, WESTERN
Electro Sales Associates
Allison Park, PA
(412) 487-3801

PENNSYLVANIA, EASTERN
Sci-Rep, Inc.
Pennsauken, NJ
(609) 662-5222
Telex 710-892-1297 SCI-REP PESN

PUERTO RICO
Data I/O
Nashua, NH
(603) 889-8511
Telex 943431

RHODE ISLAND
Data I/O
Nashua, NH
(603) 889-8511
Telex 943431

SOUTH CAROLINA

Pen Tech Associates
Greensboro, NC
(919) 852-6000

SOUTH DAKOTA

Torkelson Associates
Minneapolis, MN
(612) 835-2414
Telex 910-576-2740

TENNESSEE

Pen Tech Associates
Marietta, GA
(404) 424-1931

Huntsville, AL
(205) 881-9298

TEXAS

Testech, Inc.
Richardson, TX
(214) 644-5010

Houston, TX
(713) 956-0837

Austin, TX
(512) 327-7033

TEXAS, EL PASO CO.

Zeus Electronics, Inc.
Albuquerque, NM
(505) 842-6633
1-800-528-4512

UTAH

Zeus Electronics, Inc.
Salt Lake City, UT
(801) 534-0500
1-800-534-0503

VERMONT

Data I/O
Nashua, NH
(603) 889-8511
Telex 943431

VIRGINIA

Sci-Rep, Inc.
Fairfax, VA
(703) 385-0600
Telex 710-833-0361

WASHINGTON

Northwest T & M
Redmond, WA
(206) 881-8857

WASHINGTON, TRI-CITIES, VANCOUVER, SPOKANE

Northwest T & M
Beaverton, OR
(503) 646-9966
1-800-247-5700
Telex 910-467-8775

WASHINGTON D.C.

Sci-Rep, Inc.
Fairfax, VA
(703) 385-0600
Telex 710-833-0361

WEST VIRGINIA

Electro Sales Associates
Allison Park, PA
(412) 487-3801

WISCONSIN

Torkelson Associates
Waukesha, WI
(414) 784-7736

WYOMING

Zeus Electronics, Inc.
Denver, CO
(303) 321-4246
1-800-521-4512

U.S. SERVICE CENTERS

Redmond, WA	(206) 881-6444
Santa Clara, CA	(408) 727-0659
Costa Mesa, CA	(714) 662-2498
Nashua, NH	(603) 889-8513
Richardson, TX	(214) 235-0044

International Sales Offices and Representatives

EUROPE

Data I/O Europe
World Trade Center
Strawinskylaan 633
1077 XX Amsterdam
The Netherlands
(20) 622866
Telex 16616 DATIO NL

FEDERAL REPUBLIC OF GERMANY

Data I/O Germany GmbH
Bahnhofstrasse 3
D-6453 Seligenstadt
Federal Republic of Germany
(6182) 3088/89
Telex 4184962 DATA D

JAPAN

Data I/O Japan Company, Ltd.
Ginza Orient Bldg. 6F
8-9-13, Ginza, Chuo-ku
Tokyo 104 Japan
(03) 574-0211
Telex 2522685 DATAIO J

AUSTRALIA

Warburton Franki
Adelaide
3567333
Telex Warfran AA82579

Warburton Franki
Brisbane
(07) 277-0222
Telex Warfran 41052

Warburton Franki
Melbourne
(03) 795 9011
Telex Warfran AA31370

Warburton Franki
Newcastle
049 528722

Warburton Franki
Sydney
(02) 648-1711
Telex AA 22265

Warburton Franki
Perth
(09) 277-7000
Telex Warfran 92908

Warburton Franki
Corporate Office
Lidcombe, N.S.W. 2141
(02) 647-2266
Telex AA21299

AUSTRIA

Ing. Ernst Steiner
A-1130 Wien
(222) 827474
Telex 135026 ES A

BELGIUM

Simac Electronics
B-1000 Brussels
(2) 2192451
Telex 23662 SIMEIP B

BRAZIL

Cosele
Instrumentos Eletronicos Ltda.
01301-Sao Paulo-SP
(11) 255-1733 or 256-7421 or 231-5548
Telex (011) 30869 CSEL-BR

Cosele
Rua Sacadura Cabral, 120
Rio De Janeiro - RJ
(21) 283-2036
Telex (021) 33019

CANADA

Allan Crawford Associates
Mississauga, Ontario
(416) 890-2010
Telex 06 968769

Allan Crawford Associates
Ottawa, Ontario K1Z5L3
(613) 722-7682
Telex 053 3600

Allan Crawford Associates
St. Laurent, P.Q. H4T 1E7 (Montreal)
(514) 731-8564
Telex 05-824944

Allan Crawford Associates
Calgary, Alberta T2E 6Z5
(403) 230-1341
Telex 03 821186

Allan Crawford Associates
Burnaby, B.C. V5C6A7
(604) 294-1326
Telex 04 54247

CHINA

Dorado Company
Seattle, WA 98104
(206) 583-0000
Telex 329473 (Burgess Sea)
880212 (DORADO CO UD)

Dorado Company
Tsuen Wan, N.T. Hong Kong
0-4986401/2
Telex 51086 (FEDER HX)

Dorado Company
Beijing.
89-0621, ext. 1324

Dorado Company
c/o China Great Wall Industry Corp.
15 Zhong Shan Road (E1)
Shanghai,
219103 or 210464
Telex 33175 (CGWSH CN)

DENMARK

ITT Multikomponent A/S
DK-2600 Glostrup
(2) 451822
Telex 33355 ITT DK

FINLAND

Instrumentarium Elektroniikka
SF-02631 Espoo 63
(0) 5281
Telex 124426 HAVUL SF

FRANCE

M.B. Electronique
F-78530, Buc
(3) 9568131
Telex MB 695414 F

GERMANY, FEDERAL REPUBLIC OF

Instrumatic Electronic GmbH
D-8032 Graefelfing
(89) 852063
Telex 524298 INSTR D

GREECE

Eltronics Ltd.
GR-106 75 Athens 139
(1) 7249511 thru 15 or 7210669
Telex 216589 DARX GR

HONG KONG

Eurotherm (Far East) Ltd.
Aberdeen
5-546391
Telex 72449 EFELD HX

INDIA

Transmarketing Private Ltd.
Bombay 40018
022 4921874, 4920320 or 4926044
Telex 011 73724 TMPL IN

Bangalore
560-046

ISRAEL

Telsys Ltd.
Tel-Aviv 69010
(3) 494891 thru 5 or
(3) 494881 and 2
Telex 32392 TSEE IL

ITALY

Sistrel SPA
I-00143 Roma
(6) 5915551
Telex 680356 SISTREL I

Sistrel SPA
I-20092 Cinisello Balsamo (Mi)
(2) 6120129/6181893
Telex 334643 SISTREL I

KOREA

Elcom Systems Inc.
Seoul ZIP135
(2) 555-5222 or 557-3836
Telex K25227

MALAYSIA

GEA Technology PTE., Ltd.
Singapore 0511
65 2729412
Telex RS 37162

MEXICO

Christensen, S.A.
06470-Mexico, D.F.
546-25-95, 546-29-55
Telex 017-75612 Mycome

NETHERLANDS

Simac Electronics
5503HR Veldhoven
(40) 533725
Telex 51037 SIMAC NL

NEW ZEALAND

Warburton Franki, Ltd.
Auckland
(9) 504-458
Telex Warfran NZ 3824

NORWAY

Teleinstrument A/S
N-1371 Asker
(2) 789460
Telex 72919 TEL IN N

PORTUGAL

Decada Espectral
P-1495 Lisboa
(1) 2103420
Telex 15515 ESPEC P

SINGAPORE

GEA Technology PTE., Ltd.
(65) 2729412
Telex RS 37162

SOUTH AFRICA

Electronic Building Elements (PTY), Ltd. (EBE)
Pretoria 0001
(12) 46-9221/7
Telex 3 22786B

SPAIN

Instrumatic Espanola S.A.
Madrid-16
(1) 4558112
Telex 46277 ITIC E

Instrumatic Espanola
Malaga 5
52 213898
Telex 77131 HAFN E

SWEDEN

Macrotek AB
S-162 11 Valingby
(8) 870190
Telex 12543 MATEK S

SWITZERLAND

Instrumatic SA
CH-1207 Geneve
(22) 360830
Telex 28667 INSR CH

Instrumatic AG
CH-8800 Thalwil/ZH
(1) 7231410
Telex 826801 INBC CH

TAIWAN

Sertek International, Inc.
Taipei, 104, R.O.C.
(2) 713-5435
Telex 19162 MULTI IC or
23756 MULTI IC

THAILAND

Dynamic Supply Engineering R.O.P.
Bangkok 11
(2) 3914434, 39228532
Telex 82455 DYNASUP TH

TURKEY

Data I/O Europe
(20) 622866

UNITED KINGDOM

Microsystem Services
High Wycombe
(494) 41661
Telex 837187 MICSYS G

SOFTWARE UPDATE SERVICE

Data I/O offers you a 12-month renewable software update service that provides the best in software support. This software update service includes:

- **Software Updates** — When new devices are supported by Data I/O software, an updated copy of the software is automatically mailed to subscribers.
- **Application Notes** — Unique, special, and informative applications of the software are shared with subscribers. Printed application notes are mailed directly to subscribers.
- **Bug Fixes** — Should any software bugs be discovered, an updated copy of the software is mailed directly to subscribers.

HAVE YOU PURCHASED THE DATA I/O 12-MONTH SOFTWARE UPDATE SERVICE?

To purchase the Data I/O Software Update Service, contact your local U.S. Sales Office or Representative listed in the back of this manual, or attach a completed purchase order to the User Registration card.

OUTSIDE THE U.S.

Please contact your local International Sales Office or Representative, listed in the back of this manual.

(DETACH HERE)

(DETACH HERE)

USER REGISTRATION

Please fill in this card to register your software. If you are in the U.S., tear off and send the card to the mailing address printed on the back.

Outside the U.S., please fill in this card and send it in an envelope addressed to your local Data I/O International Sales Office or Representative.

User's Name _____

Department _____

Company _____

Address _____

_____

_____

Phone _____

*Data I/O Software Product _____

_____

*Software Version Number _____

*Operating System _____

*Serial Number _____

Date Received _____

***NOTE: Please make sure you list your Data I/O software product, its version number, the operating system it runs on, serial number, and the date you received the product. This information (except the date received) is located on the disk or tape label.**

☐ I would like more information about other Data I/O product update services.

☐ I would like to purchase the 12-month Software Update Service. Purchase Order attached.

☐ Does your company own Data I/O programming equipment? Check the box if yes!

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO. 41 REDMOND, WA 98052

POSTAGE WILL BE PAID BY ADDRESSEE

DATA I/O

User Registration/Software Update Service
Order Administration
10525 Willows Road N.E.
P.O. Box 97046
Redmond, WA 98073-9746

090-0040-001

ABEL 2.0
SYSTEM SPECIFIC INFORMATION

MS-DOS

090-0240-001

ABEL 2.0
SYSTEM SPECIFIC INFORMATION

MS-DOS

May 1986 405903

This document contains the latest information available at the time of preparation. Therefore, it may contain descriptions of functions not implemented at manual distribution time.

FutureNet Corporation provides this manual "as is," without warranty of any kind, either expressed or implied. FutureNet reserves the right to make improvements and/or changes in this document or in the product and/or program(s) described herein at any time. Information on these changes will be incorporated in new editions of this publication.

FutureNet Corporation
9310 Topanga Canyon Boulevard
Chatsworth, California 91311-5728 USA
(818) 700-0691
TWX: 910-494-2681

Copyright 1986, FutureNet Corporation.
No part of this manual may be copied or reproduced in any form or by any means without the prior written consent of FutureNet Corporation.

IBM is a registered trademark of International Business Machines Corporation.
MS-DOS and EDLIN are trademarks of Microsoft Corporation.

Table of Contents

1.	**Installing ABEL on a Dual-Drive PC**	**1-1**
1.1	The ABEL Disks	1-1
1.2	Creating Your ABEL Program Disk	1-2
1.3	Creating Your Working Disk	1-3
1.4	Setting the Path and Default Drive	1-3
2.	**Installing ABEL on an IBM-XT or -AT**	**2-1**
3.	**Large-Memory Versions of ABEL Programs**	**3-1**
4.	**Using the Personal Silicon Foundry Menu**	**4-1**
4.1	Accessing the Menu	4-1
4.2	Schematic Editing Function	4-2
4.3	Schematic Extract Function	4-3
4.4	Text Editing Function	4-4
4.5	Using ABEL Functions	4-4
4.6	Using Document and IFLDOC Functions	4-8
4.7	Using Simulate Functions	4-9
4.8	Using PLDtest Functions	4-9
4.9	Using the Terminal Emulator Utility	4-9
4.9.1	Example 1: JEDEC File Transfer, PLDS to Computer	4-12
4.9.2	Example 2: JEDEC File Transfer, Computer to PLDS	4-13
4.10	Loading a Previously Saved PSF Configuration	4-16
4.11	Saving a Specific PSF Configuration	4-17

1. Installing ABEL on a Dual-Drive PC

To run ABEL with MS-DOS/PC-DOS operating systems, you must have:

- MS-DOS/PC-DOS 2.0 or greater
- At least 256K of RAM

Before doing anything with ABEL, you should make backups of all disks. This protects you against accidental erasure of files. Refer to the operating system manual to make your backup copies of the ABEL disks.

When you have created backup copies of the ABEL files, put the original ABEL disks in a safe place and use the copies.

1.1 The ABEL Disks

ABEL is supplied on four floppy disks: the Utility disk, the Large Memory disk, the Language Processor disk, and the Design Examples disk. To use ABEL on your dual-drive PC, you'll need to copy the ABEL programs to a disk and several of the files from the Language Processor disk to a working disk.

1.2 Creating Your ABEL Program Disk

Your ABEL program disk will contain the ABEL modules used to process a logic design contained in an ABEL source file. ABEL programs can be categorized as small-memory models and large-memory models. See the large-memory version of ABEL programs which appears later in this guide.

We recommend that you copy the small-memory versions of the programs initially to your ABEL program disk, since they require significantly less processing time. You can then substitute a large-memory version of a particular program if the Memory Overflowerror condition occurs during the processing of a particular design. If you have a mix of small and large versions on one disk, be sure to change ABEL.BAT accordingly.

Using DOS, copy the following files from the Language Processor disk to a formatted non-system disk (using a non-system disk provides an additional 25K bytes of space on the disk for the substitution of large-memory modules, although the disk is not bootable):

PARSE.EXE
TRANSFOR.EXE
REDUCE.EXE
FUSEMAP.EXE
SIMULATE.EXE
DOCUMENT.EXE
ABEL.BAT
COMMAND.COM (from your DOS disk)

This provides you with your ABEL program disk. Use this disk in drive A.

1.3 Creating Your Working Disk

When using ABEL, the working disk is used in drive B. To create the working disk, copy the following files from your copy of the supplied Utility disk to a formatted system disk:

ABELLIB.DEV (device file library)
PSF.CFN (configuration file)
PSF.EXE (Personal Silicon Foundry-PSF)
PSF.STR (PSF data string file)

If you wish to use PC-Write for your text editor, copy ED.EXE from the PC-Write disk to the sytem disk described above.

ABELLIB.DEV contains files for all supported PLDs. If you are going to use only a few logic devices, and don't need the entire device library, you can copy selected files to your working disk from the individual device files supplied on the Examples disk.

1.4 Setting the Path and Default Drive

To use ABEL, insert the program disk in drive A and the working disk in drive B. At the DOS prompt, set the path and default drive as follows:

```
PATH = A:\
B:
```

(You can insert these commands in the AUTOEXEC.BAT file of your boot disk.)

2. Installing ABEL on an IBM-XT or -AT

To run ABEL with MS-DOS/PC-DOS operating systems, you must have:

- MS-DOS/PC-DOS 2.0 or greater
- At least 256K of RAM

Installation of the ABEL program on a hard drive XT/AT consists of copying files from the supplied floppy disks onto the hard drive. You can install the files into your directory structure, or you can use the following procedure to install ABEL in specific directories.

If you copy the files to your own directory structure, be sure to do the following:

- Place the *.EXE files in the search path.
- Place the device file library (abellib.dev) and configuration (*.cfn, *.str) files in a directory that is referenced by the ABELDEV environment variable (using the SET command).

Proceed with the installation of ABEL as follows:

1. Make directories on the hard disk using these command lines.

```
A>md c:\dataio
A>md c:\dataio\lib
A>md c:\dataio\abel
```

2. Insert the ABEL Language Processor disk into drive A. Enter the following commands to copy the ABEL Language Processor files onto the hard disk:

```
A>copy *.* c:\dataio
```

3. Remove the Language Processor disk from drive A and insert the ABEL Large Memory disk. Use the following command to copy the large memory programs to the hard disk:

```
A>copy *.* c:\dataio
```

4. Remove the Large Memory disk from drive A and insert the ABEL Design Examples disk. Use the following command to copy the example files to the hard disk:

```
A>copy example\*.* c:\dataio\abel
```

5. Remove the Design Examples disk from drive A and insert the ABEL Utilities disk into drive A. Enter the following commands to copy the ABEL Utilities onto the hard disk:

```
A>copy *.exe c:\dataio
A>copy *.com c:\dataio
A>copy *.cfn c:\dataio\lib
A>copy *.str c:\dataio\lib
A>copy *.dev c:\dataio\lib
```

6. Several ABEL functions require a text editor. At this point you must install the text editor of your choice, if not already done.

   You must use a text editor that produces a standard ASCII file. When using WordStar, use the non-document mode.

   The text editor PC-Write is included with the ABEL package. If you choose to use PC-Write, insert the PC-Write disk in drive A and use the following commands t install the text editor on the hard disk:

   ```
   A>md c:\dataio\pcwrite
   A>copy *.* c:\dataio\pcwrite
   A>copy ed.exe c:\dataio
   ```

   Note that this installation of the text editor provides onl the basic editing capability of PC-Write, which is sufficient for use with ABEL. To add the full word-processing capabilities, refer to the PC-Write manual.

7. Use your editor to modify or create an AUTOEXEC.BAT file. For example, using PC-Write, type the following command:

   ```
   A>c:\dataio\ed c:\autoexec.bat
   ```

8. Then add the following instructions to the batch file:

```
path c:\com;c:\dataio
set abeldev=c:\dataio\lib
set comspec=c:\command.com
mode com1:9600,n,7,1
```

If you have installed another text editor in a directory other than "com" or "dataio", you may wish to put that directory in the search path. For example, if you have copied your text editor into a new directory named editor, then use the following path command:

```
path c:\com;c:\dataio;c:\editor
```

9. Modify your CONFIG.SYS file, or create a new one, and add the following to specify eight buffers, 15 open file handles, and ANSI.SYS:

```
buffers=8
files=15
device=c:\com\ansi.sys
```

The ABEL Language Processor will run faster with the buffers set to 40 in the CONFIG.SYS file; however, more RAM is used.

The device configuration command specifies ANSI.SYS to provide the extended screen and keyboard functions for use of the Personal Silicon Foundry (PSF) feature of the ABEL package.

This completes installation of the ABEL software in the IBM-XT or AT.

3. Large-Memory Versions of ABEL Programs

Several of the executable files supplied with the ABEL package use all available RAM to process designs that contain many terms. These programs are named BIG???.EXE, where ??? is name of the smaller version of the program and should be used whenever ABEL indicates a Memory Overflow error.

The large memory versions may be used in the event an out-of-memory (memory overflow) condition occurs during the execution of the Language Processor. The smaller versions of the ABEL programs operate at a higher speed than the large memory versions, but use a maximum of only 64K of RAM. Because of the large number of terms in designs for some FPLAs and large PLDs, more than 64K of RAM may be required during processing. In these instances, the large-memory versions of the programs should be used.

A batch file is provided that can be used to call the large-memory ABEL programs when processing large designs. The batch file, named "bigabel.bat", is provided for use with an IBM-AT or -XT. When running ABEL with designs that require large amounts of RAM, use the command:

```
A>bigabel <filename>
```

The processing times for the large memory versions of the Language Processor programs are noticeably longer than the processing times of the smaller versions of the same programs. For this reason, you may wish to modify the batch file to include only those large memory programs you need. For example, if the Memory Overflow error occurs only during REDUCE, you can modify the batch file to execute BIGRED.EXE, but only the small memory versions of the other ABEL modules.

If you are using a dual drive PC, there is not sufficient space on the floppy disk for all large model programs to be used at one time. If you encounter a Memory Overflow error while running ABEL on the PC, modify the "abel.bat" file to replace only those programs that need more memory.

4. Using the Personal Silicon Foundry (PSF) Menu

The ABEL Utilities disk contains a program named "psf.exe" (the Personal Silicon Foundry) and a pair of files named "psf.cfn" (the configuration file) and "psf.str" (a data file required for the use of psf.exe). After these files are copied to your system as described in the above procedure, you can, as an option, access and use the Personal Silicon Foundry menu to execute ABEL and related functions.

Personal Silicon Foundry is for the IBM-PC/-AT/-XT and compatibles. To run "psf.exe" you must have "ansi.sys" in the DOS configuration file (config.sys).

4.1 Accessing the Menu

When ABEL is installed in accordance with the above instructions, the path is set so that entering

```
psf
```

on the command line will access the PSF menu.

Note that the Schematic Editing selection is highlighted by a solid bar to indicate that the F9 and F10 keys can be used to access this function. You can use the up-arrow and down-arrow keys to move this bar and highlight any of the listed selections. You can then use F9 and F10 to access submenus or execute the highlighted function.

To access the submenu for Schematic Editing, Schematic Extract, Text Editing, ABEL, Document, Simulate, or PLDtest, press F9 with the highlight bar placed over the selection. To execute the highlighted function (using previously selected parameters), press F10.

4.2 Schematic Editing Function

If the DASH or STRIDES schematic design and editing system is installed in your system, you can select this function from the PSF main menu to perform schematic diagram generation and editing. For details regarding the schematic editor, refer to your DASH/STRIDES documentation.

When selecting the Schematic Editing function, you can press F9 or F10, depending on the need to specify different parameters from those last entered or previously saved. If none of the schematic editing parameters (editor name, symbol library, and drawing file) need to be changed, press F10 and the system will automatically access the desired schematic file ready for editing.

If you need to alter the schematic editing parameters, proceed as follows:

1. Press F9 and a submenu appears on the display, with the Editor field highlighted. At this point you can change any of the three schematic editor parameters: editor name, symbol library, and drawing file.

2. To change a parameter, use the up-arrow and down-arrow keys to highlight the parameter then press F10. Note that the name of the parameter changes to upper case letters signifying that the parameter can be entered. Press the Return key to complete the entry.

3. When all changes are complete, move the highlight down to the Execute selection and press F10 to begin the editing session, or press F9 to return to the main PSF menu.

Unless you save (see Saving a Specific PSF Configuration) your parameters, the selections made on the submenu will remain in effect only until the system is powered down or new parameters are loaded (see Loading a Previously Saved PSF Configuration).

4.3 Schematic Extract Function

If DASH-ABEL (the schematic capture interface system) is installed in your system, you can select this function from the PSF main menu. As in the Schematic Editing function, you can press F9 or F10, depending on the need to specify different parameters from those last entered or previously saved.

If none of the DASH-ABEL parameters, that is, function library, drawing file, and -S (suppress abbreviating device part numbers) are to be changed, press F10 and the system will automatically process the specified drawing file to the ABEL source file.

If you need to alter the DASH-ABEL parameters, proceed as described above for the Schematic Editing function, to enter the desired parameters.

As with the Schematic Editing function, the selections made on the submenu will remain in effect only until the system is powered down, or new parameters are loaded (see Loading Configurations Files).

4.4 Text Editing Function

You can also select your text editor from the PSF main menu. You can specify a different file from that last accessed, or specify an alternate data file, or simply access the file that was last specified, by means of the F9 and F10 keys. Note that two files can be specified (ABEL file and Altn file) and saved with the configuration, although only the file with the asterisk will be accessed when the Text Editing function is executed.

This feature allows you to keep track of two data files that may be required for a particular design, such as an ABEL source file and include file, and eliminates the need to re-enter the name of each file when alternating between the two. To specify the desired file for editing, move the asterisk so that it is adjacent to the file (use the up/down arrow and F10 keys to do this) and then move the highlight bar to Execute and press F10.

4.5 Using ABEL Functions

If you wish to run ABEL with parameters previously loaded or specified in the PSF menu, there is no need to access the ABEL submenu, simply move the highlight bar down to the ABEL line and press F10. This invokes the ABEL language processor with the existing parameters and the display then shows the processing operation described in the ABEL User's Guide.

To specify other parameters for the ABEL processing operation, press F9 to access the ABEL submenu. Use the up/down/left/right arrow keys to select the desired parameters. Each of the parameters is described in detail in Section 4 of the ABEL User's Guide and a summary of parameter selections offered by the PSF ABEL submenu is given in Table 1.

The default (DFLT) selections shown in the ABEL submenu allow the values specified by the Flag Statement (in the ABEL source file) to take precedence. Refer to Section 3 of the Language Reference manual for a description of the Flag Statement.

Table 1. ABEL Options with the PSF Menu

PARAMETER	MEANING
Input File Name	This is the name of the ABEL source file to be processed.
Include File Path	This is the directory path (-Hpath) where the Parse program of ABEL is to look for files to be included with the source file (by means of the @INCLUDE directive).
JEDEC File Path	This is the drive and path (-Jpath) that indicates where the Fusemap program of ABEL is to write the programmer load file (JEDEC file).
Arguments	This is the name of the argument (-Aarg) substituted for the dummy arguments in the source file by the Parse program.
Reduce	This is the level of logic reduction (-Rn) provided by the Reduce program. The choices are: default, no reduction, simple reduction, and simple & PRESTO reduction by pin. Refer to Section 4 of the User's Guide.

PARAMETER	MEANING
Checksum	This is checksum selection used by the Fusemap program. The selections correspond to the -Cn parameter described in Section 4 of the User's Guide.
Fusemap Output	This specifies the format of the programmer load file for the Fusemap program. The selections correspond to the -Dn parameter described in Section 4 of the User's Guide.
Connect Fuses	This specifies whether or not unused fuses in an IFL or FPLA OR array will be disconnected or left connected. The selections correspond to the -Kq parameter described in Section 4 of the User's Guide.
List File	Two selections can be made here; one to create or suppress the list file produced by the Parse program, and the other to select normal list file, the writing of expanded source, and the listing of directives. The selections correspond to the -L, -E, and -P parameters described in Section 4 of the User's Guide.
Trace	This is the trace level for the simulate operation. The selections correspond to the -Tn parameter described in Section 4 of the User's Guide.

PARAMETER	MEANING
Don't Care	This specifies the logic level assigned to don't care values contained in the test vectors of the ABEL source file. The selection corresponds to the -Xn parameter described in Section 4 of the User's Guide.
Hi Z	This specifies the logic level assigned to high-impedance values contained in the test vectors of the ABEL source file. The selection corresponds to the -Zn parameter described in Section 4 of the User's Guide.
Breakpoints	This specifies the breakpoints in the simulation process. Refer to the -Bn parameter described in Section 4 of the User's Guide.
Waveform	This specifies the device pins to appear on the trace level 4 waveform output. Refer to the -Wn parameter described in Section 4 of the User's Guide.
Equations	This specifies the equations to be listed by the Documentation program. Refer to the -Qxyz parameter described in Section 4 of the User's Guide.
Chip Diagram	This specifies to list or not list the diagram of the programmed part. Refer to the -G parameter described in Section 4 of the User's Guide.

PARAMETER	MEANING
Vectors	This specifies to list or not list the test vectors from the source file. Refer to the -V parameter described in Section 4 of the User's Guide.
Symbols	This specifies listing the symbol table derived from the source file. Refer to the -S parameter described in Section 4 of the User's Guide.
Fuse Map	This controls the listing of fusemaps and terms. Refer to the -F parameter described in Section 4 of the User's Guide.

4.6 Using Document and IFLDOC Functions

If you wish to run the ABEL document function with parameters previously loaded or specified in the PSF menu, there is no need to access the Document submenu. Simply move the highlight bar down to the Document line and press F10. This invokes the Document program of the ABEL language processor.

To specify other parameters for the document operation, press F9 to access the Document submenu. Use the cursor movement keys to select the desired parameters. Each of the parameters is described in detail in Section 4 of the ABEL User's Guide and also in Table 1 of this chapter. Selections made to the Document and IFLDOC submenu do not affect the similarly named parameter selections that appear on the previously described ABEL submenu. The Document submenu also allows you to select parameters for and execute the IFLDOC utility. This program is described in Section 6 of the ABEL User's Guide.

4.7 Using Simulate Functions

If you wish to run the ABEL simulate function with parameters previously loaded or specified in the PSF menu, simply move the highlight bar down to the Document line and press F10. This invokes the Simulate program of the ABEL language processor.

To specify other parameters for the simulate operation, press F9 to access the Simulate submenu. Use the cusor control keys to select the desired parameters. Each of the parameters is described in detail in Section 4 of the ABEL User's Guide and also in Table 1 of this chapter.

Selections made to the Simulate submenu do not affect the similarly named parameter selections that appear on the previously described ABEL submenu.

4.8 Using PLDtest Functions

If you have PLDtest installed on your system, you can use the highlight bar to make the PLDtest selection of the PSF main menu, and the PLDtest submenu to invoke and control PLDtest operations. Refer to your PLDtest manual for additional details.

4.9 Using the Terminal Emulator Utility

This utility configures your computer so that it emulates a data terminal. When using your computer as a data terminal, you can perform file transfers to and from mainframe computers, Data I/O programmers, and other personal computers via the RS-232 COM1 port (full or half duplex).

If you used these instructions to install ABEL on your system, set the COM1 port to 9600 baud, no parity, eight data bits, and one stop bit. To alter the communications protocol of the COM1 port, use the DOS mode command as explained in the DOS manual.

To use the Terminal Emulator Utility, use the highlight bar to select the Terminal line on the PSF main menu. Press F10 and the menu appears on the display. Refer to Table 2 for an explanation of each TERM function.

Table 2. TERM Commands and Function

COMMAND	FUNCTION
F1	Closes any open files and then exits to DOS.
F2	Stops file I/O transfer and closes any open files.
F3	Displays the help screen, which consists of the list of commands and a brief description.
F4	Receive file: Permits user to create a file name for a file to be received and saved to disk. Note that files are written to disk exactly as received. Specify a path for the file (if required); type filename (up to eight characters plus dot and three character extension) in status block as requested and press the Return key.

COMMAND	FUNCTION
F5	Transmit file: Permits the user to specify a file name for a file to be transmitted from the computer to equipment connected to COM1: port. Specify a path for the file (if required); type filename (up to eight characters plus dot and three character extension) in status block as requested and press the Return key.
F6	Binary or text mode of file transmission set by toggling between them; default is Binary mode; Text mode suppresses (strips out) control-Z, control-C, line feeds, and nulls from files being sent from computer to COM1: port via transmit file (F5) function. Do not use text mode when transmitting JEDEC files to PLDS. See TERM menu status block for current mode.
F7	Full or half duplex transfer mode set for file transmit/receive (F5 and F4) functions -- default is Full duplex. See TERM menu status block for current mode.

To assist you in using the Terminal Emulator Utility, two examples are provided. Each example shows how to transfer a JEDEC formatted file from your computer to a Data I/O Model 29 programmer (with a LogicPak and P/T Adapter, called PLDS Programmable Logic Development System) or vice versa.

The following examples assume that the equipment is connected correctly and that the COM1: port is appropriately configured. The TERM utility must be installed in the computer.

4.9.1 Example 1: JEDEC File Transfer, PLDS to Computer

In this example, a JEDEC format device file must be loaded into the programmer RAM and be ready for down loading to the computer.

1. To run the TERM program, type:

   term<return>

   Your computer is now a terminal.

2. Set the Model 29 to full duplex (echo) operation by entering the following key sequence from the programmer front panel:

   SELECT C E START 0 START

3. Enter the PLDS terminal mode (TRC) from the Model 29 programmer front panel by entering:

   SELECT E 1 START

4. Prepare TERM to receive a file by selecting the full duplex mode option from the menu, only if necessary, since TERM defaults to full duplex at start up):

   F7 (toggles between full and half)

5. Select the Create disk file function to receive the transmitted file. (You can specify a disk drive and path prior to filename, if required.)

   F4 filename<return>

6. Instruct the PLDS to transmit the JEDEC file by pressing (at the computer keyboard):

   C

   You should see the device file scroll down the screen as it is being written into the disk file.

7. Using TERM's Terminate file transfer function, close the file transmission by pressing:

   F2

This example is concluded and the JEDEC file transfer from the PLDS to your computer is complete. There are other step sequences for down loading a file from the programmer to the computer, using TERM. The preceding method is perhaps the most direct, requiring the least number of steps.

4.9.2 Example 2: JEDEC Transfer, Computer to PLDS

In this example, a JEDEC format device file must be created and saved to disk. If you are performing these examples in sequence, you can transfer the same device file from example 1 back to the PLDS. Make sure that the COM1: serial port is correctly configured.

1. Run the TERM program by typing the following characters from the computer keyboard:

   term<return>

2. Establish that the Model 29 is in full duplex mode. If it isn't, enter the following key sequences from the programmer front panel:

   SELECT C E START 0 START

3. Enter the PLDS terminal mode from the Model 29 programmer front panel by entering:

   SELECT E 1 START

   The PLDS menu will be displayed, including the PLDS command prompt. Most of this menu will overwrite the TERM program menu on the screen except for the status block. The TERM status block and PLDS command prompt inform you that you can communicate with both the TERM program and the PLDS equipment through the terminal emulator.

4. At the PLDS command prompt (from the computer), type:

   l xxxx<return>

   where l is the PLDS menu selection which permits specification of device-to-be-programmed family and pinout code. The xxxx represents, and is replaced by, the four-digit device family and pinout code number that you supply.

5. Prepare the PLDS to receive a JEDEC file. At the PLDS command prompt (from the computer), type:

   B

6. To recall the complete TERM menu, press:

   F3

   The TERM menu will be displayed. (It is not necessary to recall this menu if you can remember the functions keys.)

7. Prepare TERM to transmit a file by specifying the appropriate transmission mode (use binary mode to transmit JEDEC files) from the TERM menu:

   F6 (toggles between binary and text modes)

   See the status field at the bottom of the screen - TERM defaults to binary mode at start-up.)

8. Set TERM for full duplex, only if necessary, since TERM defaults to full duplex at start up:

   F7 (toggles between full and half duplex)

9. Select the Send disk file function from the TERM menu:

   F5 filename<return>

   to transmit the file. The file being transferred must be for the same device family and pinout code as that entered previously from the PLDS menu. (You can specify a disk drive and path prior to filename if required.)

   When file transfer is complete, the PLDS command prompt will be displayed.

10. Using TERMs terminate file transfer function, close the file by pressing:

    F2

This concludes this example and the JEDEC file transfer from your computer to the PLDS.

The examples provided describe only two of the possibilities for using TERM. Notice that both examples make use of features or capabilities that originate from PLDS menu option selections and programmer/pak select code functions.

When your programmer is set in Terminal Remote Control and TERM has converted your computer into a terminal, all of these programmer/pak features or capabilities are accessible. Consult the programmer and/or LogicPak manual for more information pertaining to the use of these features and capabilities.

4.10 Loading a Previously Saved PSF Configuration

Instead of accessing each submenu and altering the parameters to accommodate each specific situation, you can load a configuration file that you previously created with the Save Configuration file feature (described below). To load a configuration file, move the highlight bar down to the Load Configuration line of the PSF main menu and press F10. Enter the name (and path if necessary) of the configuration file and press the Return key. PSF.CFN is loaded automatically when the menu is invoked; if a different configuration file is loaded, the name of the file will appear on the main menu.

4.11 Saving a Specific PSF Configuration

This main menu selection allows you to save the parameters that presently exist in each of the PSF submenus to a named file. To use this feature, move the highlight bar down to the Save Configuration line and press F10. Enter the name (and path if necessary) of the configuration file to which the parameters are to be saved.